GENERADOR DE COMPILADORES BASADO EN ANALIZADORES ASCENDENTES

Memoria del Proyecto de Final de Carrera
Ingeniería Informática realizado por Laia Felip Molina
y dirigido por Xavier Sánchez Pujades

Bellaterra, 18 de Septiembre de 2009

Escola Tècnica Superior d'Enginyeria

El abajo firmante Xavier Sánchez Pujades

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

CERTIFICA:

Que el trabajo al que corresponde esta memoria ha sido realizado bajo su dirección por Laia Felip Molina

Y por tal que conste firma la presente.

Firmado: Xavier Sánchez Pujades

Bellaterra, 18 de Septiembre de 2009

A mis padres por enseñarme que el esfuerzo es el camino para lograr tus objetivos.

A mis tíos, por su ayuda y apoyo incondicional.

A Dani, por no dejar de creer en mí.

A Kenny, por su compañía.

TABLA DE CONTENIDO

1. INTRODUCCIÓN 1

1.1. OBJETIVOS 1
1.2. MOTIVACIONES 2
1.3. ESTADO DEL ARTE 2
1.4. CLASIFICACIÓN DE LA MEMORIA 6

2. PLANIFICACIÓN 7

2.1. PLANIFICACIÓN INICIAL 7
2.2. PLANIFICACIÓN FINAL 9
2.3. ESTUDIO DE VIABILIDAD 12

3. UN COMPILADOR 13

3.1. ANALIZADOR LÉXICO 14

3.2. ANALIZADOR SINTÁCTICO 16

3.2.1. GRAMÁTICAS LIBRES DE CONTEXTO 16
3.2.2. DERIVACIÓN 17
3.2.3. ÁRBOLES SINTÁCTICOS 18
3.2.4. MÉTODOS DE ANÁLISIS SINTACTICOS 19
3.2.5. ANÁLSISIS ASCENDENTE 23

3.2.5.1. LR (K) 25

3.2.5.1.1. ANÁLISIS LR (0) 27
3.2.5.1.2. TABLA DE ANÁLISIS ASCENDENTE 31
3.2.5.1.3. INICIO DE ANÁLISIS 32
3.2.5.1.4. ANÁLISIS LR (1) 36
3.2.5.1.5. ANÁLISIS LALR (1) 43

3.3. ANALIZADOR SEMÁNTICO 46

3.3.1. GRAMÁTICA DE ATRIBUTOS 47
3.3.2. TIPOS DE ATRIBUTOS 48

3.4. GENERADOR DE CÓDIGO 50

4. DESARROLLO 51

4.1. LÉXICO 51
4.2. SINTÁCTICO 51

4.2.1. AUTÓMATA LR (1) 53

4.2.1.1. CREAR PRODUCCIÓN INICIAL 54
4.2.1.2. CREAR CLAUSURA 54
4.2.1.3. CREAR ESTADO 56
4.2.1.4. CREAR TRANSICIONES 57
4.2.1.5. VER ESTADOS 58

4.2.2. AUTÓMATA LALR (1) 62
4.2.3. TABLA DE ANÁLISIS 64
4.2.4. ANALIZADOR LALR (1) 66

4.3. AÑADIR ACCIONES SEMÁNTICAS 68

4.3.1. FUNCIÓN DE REDUCCIÓN 70
4.3.2. ATRIBUTOS HEREDADOS 75

5. CONCLUSIONES 79
6. ANEXO 81

6.1. EXPRESIONES REGULARES 81
6.2. CÁLCULO DE ANULABLES 82
6.3. CÁLCULO DE PRIMEROS 83
6.4. EJEMPLOS DEL ANALIZADOR ASCENDENTE LALR (1) 84

BIBLIOGRAFÍA 89
ÍNDICE 91

TABLA DE ILUSTRACIONES

FIGURA 2.1 PLANIFICACIÓN INICIAL Y FINAL 11

FIGURA 3.1 DEFINICIÓN DE UN COMPILADOR 13

FIGURA 3.2 FASES DE UN COMPILADOR 14

FIGURA 3.1.1 ANALIZADOR LÉXICO 15

FIGURA 3.2.1 ANALIZADOR SINTÁCTICO 16

FIGURA 3.2.2 ÁRBOL SINTÁCTICO 19

FIGURA 3.2.3 ANÁLISIS DESCENDENTE (DERIVACIÓN POR LA IZQUIERDA) 21

FIGURA 3.2.4 ANÁLISIS ASCENDENTE (DERIVACIÓN POR LA DERECHA) 23

FIGURA 3.2.5 GRAMÁTICA LR (0) 27

FIGURA 3.2.6 AUTÓMATA LR (0) 30

FIGURA 3.2.7 ESTRUCTURA DEL ANALIZADOR SINTÁCTICO 32

FIGURA 3.2.8 TABLA DE ANÁLISIS LR (0) 33

FIGURA 3.2.9 ANALIZADOR LR (0) 35

FIGURA 3.2 10 GRAMÁTICA PARA MOSTRAR TODOS LOS MÉTODOS 35

FIGURA 3.2.11 AUTÓMATA LR (0) CON CONFLICTO DESPLAZAR-REDUCIR 36

FIGURA 3.2.12 AUTÓMATA LR (1) 40

FIGURA 3.2.13 TABLA DE ANÁLISIS DEL AUTÓMATA LR (1) 41

FIGURA 3.2.14 ANALIZADOR LR (1) 42

FIGURA 3.2.15 AUTÓMATA LALR (1) 44

FIGURA 3.2.16 TABLA DE ANÁLISIS DEL AUTÓMATA LALR (1) 45

FIGURA 3.2.17 ANALIZADOR LALR (1) 45

FIGURA 3.3.1 ANALIZADOR SEMÁNTICO 46

FIGURA 3.3.2 ATRIBUTOS SINTETIZADOS 48

FIGURA 3.3.3 ATRIBUTOS HEREDADOS 49

FIGURA 4.2.1 ESTADOS DEL AUTÓMATA LR (1) 60

FIGURA 4.2 2 AUTÓMATA LR (1) EN COSEL 61

FIGURA 4.2.3 ESTADOS LALR (1) 63

FIGURA 4.2.4 AUTÓMATA LALR (1) EN COSEL 64

FIGURA 4.2.5 VECTOR DE LISTAS DE ANÁLISIS 65

FIGURA 4.3.1 PILA EN ANÁLISIS SINTÁCTICO Y SEMÁNTICO 69

FIGURA 4.3.2 AUTÓMATA LALR (1) CON ATRIBUTOS 73

FIGURA 4.3.3 ANÁLISIS LALR (1) CON ATRIBUTOS 74

Capítulo 1

INTRODUCCIÓN

1.1 OBJETIVOS

El objetivo fundamental de este proyecto consiste en crear un generador de compilador, basado en analizadores ascendentes. Como base para hacer este analizador se usará el lenguaje Cosel y el módulo com, que es un generador de compiladores basado en analizadores descendentes y que actualmente se está utilizando en las prácticas de la asignatura de Compiladores I. El nuevo generador, que tiene como entrada una gramática, ha de comprobar si es una gramática ascendente LALR (1) y analizar una cadena de entrada de símbolos usando dicha gramática.

La compilación se compone principalmente en cuatro etapas:

- **LÉXICO**
  En este apartado, únicamente se ha de aplicar un scanner, que ya existía previamente, sobre el generador de compiladores Com de Cosel. De esta manera se pueden leer todos los símbolos de la cadena de entrada.

- **SINTÁCTICO**
  Se parte de las gramáticas generadas en el modulo Com, para un análisis descendente. Con estas se pueden obtener todos los símbolos de dicha gramática, símbolos terminales y no terminales, transformaciones de BNF a producciones, producciones, anulables, primeros, siguientes, etc; usados como base para generar un analizador ascendente. De esta forma, se puede crear una autómata LALR (1), analizarlo y comprobar si la gramática de entrada pertenece a una gramática ascendente LALR (1). Finalmente, analizar una cadena de entrada de símbolos usando dicha gramática. Esta es la etapa más importante de la compilación.

- **SEMÁNTICO Y GENERACIÓN DE CÓDIGO**
  Estas etapas de la compilación, son realizadas mediante Cosel. Por este motivo, simplemente el generador Com ha de permitir la introducción de atributos y acciones semánticas. De esta manera, se pueden verificar las declaraciones de las variables y las gramáticas libres contexto.

1.2 MOTIVACIONES

Este proyecto está destinado a ser una herramienta docente para los alumnos. Por tanto, ha de ser fácil de aplicar y visualizar, cosa que no cumplen los generadores ascendentes que existen actualmente YACC y Bison. Este hecho se hace patente sobretodo en el tratamiento de atributos heredados y sintetizados, ya que usan una notación basada en *$*, que es poco legible y comprensible.

1.3 ESTADO DEL ARTE

Con la aparición del primer ordenador digital en 1946, se hizo necesario escribir secuencias de código, o programas, que darían como resultado que estas computadoras realizaran los cálculos deseados. En un primer momento, estos programas se escribían en lenguaje máquina: códigos binarios que representaban las operaciones reales de la máquina que van a efectuarse. Este tipo de lenguajes son de rápida interpretación por parte de la máquina, pero de difícil comprensión del ser humano.

A causa de la molesta y lenta escritura de los códigos binarios, esta forma de codificar pronto fue reemplazada por el lenguaje ensamblador, que mejoró extraordinariamente la velocidad y precisión con la que se podían escribir los programas, pero tenía varios defectos: aún era difícil de escribir, leer y comprender. Por otro lado, el lenguaje ensamblador dependía por completo de la máquina en particular para la que se había escrito el código. Esto significa que si un código se quería exportar a otra computadora, este se tenía que volver a escribir desde cero.

Posteriormente, aparecen los lenguajes de alto nivel. Estos lenguajes se escriben mediante instrucciones o sentencias similares a las de los lenguajes humanos. Esto facilita la escritura y comprensión del código al programador. Para que este tipo de lenguaje se pueda expresar en

diferentes máquinas, surge el concepto de compilador, un programa que traduce un lenguaje de alto nivel a otro de bajo nivel. [1]

En la actualidad existen muchos lenguajes de alto nivel, para los cuales se han creado muchas metodologías basadas en la generación de compiladores. Estas metodologías se basan en el análisis sintáctico tanto ascendente como descendente.

El analizador sintáctico que se propone en este proyecto es el analizador ascendente LALR (1). Algunos de los compiladores basados en analizadores ascendentes LALR (1) que se usan actualmente son: BISON y YACC.

- **YACC** (Yet Another Compiler-Compiler)
  Es un programa muy extendido en UNIX. YACC acepta como entrada una gramática libre de contexto, y produce como salida el autómata finito que reconoce el lenguaje generado por dicha gramática. YACC solo acepta las gramáticas LALR (1). Además, permite asociar acciones a cada regla de producción y asignar un atributo a cada símbolo de la gramática ya sea terminal o no terminal. Dado que el autómata que genera YACC está escrito en lenguaje C, todas las acciones semánticas y declaraciones necesarias, deben ser escritas en este mismo lenguaje. YACC parte de unos símbolos terminales que deben ser generados por un analizador léxico. Dicho analizador puede implementarse a partir de LEX.

  Cada regla gramatical se expresa de la siguiente forma:

```
noTerminal: componentes;
```

  donde componentes es una secuencia de cero o más terminales y no terminales. Por convención, los símbolos no terminales se escriben en minúscula y los terminales en mayúscula, las flechas se sustituyen por *":"* y cada regla debe acabar en *";"*. Se va a mostrar una correspondencia entre gramática formal y su equivalencia en notación YACC. [2]

Gramática formal	Notación YACC
E→ E+T	e: e '+' t
| T	| t
	;

Junto con los símbolos terminales y no terminales, se pueden añadir acciones semánticas en forma de código C delimitado por llaves. Por ejemplo:

```
e : e ‘+’ t {Printf(“Esto es una suma\n”);};
```

La importancia de las acciones semánticas, es asociar a los símbolos terminales y no terminales un atributo. Los nombres de los atributos son siempre los mismos y dependen únicamente de la posición que ocupa el símbolo dentro de la regla. De esta forma, el nombre *$$* se refiere al atributo de la parte izquierda de la regla, mientras que *$i* se refiere al atributo asociado al i-enésimo símbolo de la gramática *componentes*. Es decir,

- El atributo del antecedente es ***$$***.
- El atributo del primer símbolo de *componentes* es ***$1***.
- El atributo del segundo símbolo de *componentes* es ***$2.***
- Etc.

Las acciones semánticas se ejecutarán siempre que se reduzca la regla, por lo que esta se encarga de calcular un valor para *$$* en función de *$i*. Por ejemplo:

```
e: e ‘+’ t {$$=$1+$3;}
| t {$$= $1;};

t:  t ’*’ f {$$=$1+$3;}
| f {$$= $1;};

f:  NUM {$$=$1;}
```

El analizador léxico se encarga de asignar un valor a los atributos de los terminales, en este caso *NUM*.

- **BISON**

  Es un generador de analizadores sintácticos perteneciente al proyecto GNU, que convierte una descripción gramatical para una gramática independiente de contexto LALR (1), en un programa en C que analiza programas. Bison es compatible con YACC, por tanto, todas las gramáticas escritas para YACC deberían funcionar con Bison,

obteniendo el mismo resultado. Usándolo junto a LEX esta herramienta permite construir compiladores de lenguaje. [3]

Una regla gramatical de Bison tiene la misma notación que YACC.

```
exp: exp ‘+’ exp
```

Igual que ocurre con YACC, una regla gramatical puede tener una acción compuesta de sentencias en C. Cada vez que el analizador reconozca esa regla, se ejecuta la acción.

En este ejemplo, se muestra una regla que dice que una expresión puede ser la suma de dos subexpresiones:

```
exp: exp '+' exp   { $$ = $1 + $3; };
```

- **LEX / FLEX**

  Son generadores de analizadores léxicos que reciben como entrada un fichero de texto que contiene las expresiones regulares que corresponden a las unidades sintácticas del lenguaje que se va a compilar. Estos generadores crean un fichero en código C, que contiene una función llamada yylex() que implementa el analizador léxico. [2]

  Este fichero LEX/FLEX consta de tres partes: la sección de definiciones, reglas y rutinas auxiliares. Como se puede ver en el siguiente ejemplo:

```
/*Sección de definiciones*/
      #include <stdio.h> //Para usar Printf en las reglas

/*Sección de Reglas*/
      Digito [0-9]

/* Sección de rutinas auxiliares*/
      Begin  {printf(“Begin reconocido \n”);}
      End    {printf(“End reconocido \n”);}

Int main(){
      Return yylex();//Si llega al fin de la entrada termina
      }
```

1.4 CLASIFICACIÓN DE LA MEMORIA

La memoria está dividida en diferentes capítulos. A continuación, se explica brevemente cada uno de ellos.

CAPÍTULO 2: PLANIFICACIÓN

En este capítulo, se muestra la planificación inicial y final del proyecto. También se realiza el estudio de viabilidad del proyecto.

CAPÍTULO 3: UN COMPILADOR

En este apartado se explica la parte teórica del proyecto. Qué es un compilador, cuáles son sus etapas y qué realiza cada una de ellas. Sobre todo, se da especial importancia a la fase de análisis sintáctico, que es donde se usan los analizadores ascendentes. También se muestran diferentes ejemplos para su comprensión.

CAPÍTULO 4: DESARROLLO

En esta sección, se muestra como se ha realizado un generador sintáctico ascendente, basado en la parte teórica explicada en el capítulo 3. Se explican diferentes ejemplos para visualizar y demostrar su buen funcionamiento.

CAPÍTULO 5: CONCLUSIONES

En este capítulo, se detalla cuales son las conclusiones extraídas después de la elaboración del proyecto. Cuáles han sido los objetivos conseguidos y cuáles son los pendientes.

CAPÍTULO 6: ANEXO

Se adjunta información complementaria y más ejemplos, para una mejor comprensión.

BIBLIOGRAFÍA
ÍNDICE

Capítulo 2

PLANIFICACIÓN

En este capítulo, se describe la planificación inicial y final del proyecto. Se han representado mediante un diagrama, como se puede observar en la Figura 2.1. En estos diagramas, se muestra como se ha visto alterada la planificación final respecto la inicial. Esto se ha producido debido a la falta de experiencia en la planificación de proyectos y, obviamente, a causa de la aparición de imprevistos durante el desarrollo del mismo. También se encuentra el estudio de viabilidad correspondiente al proyecto realizado.

2.1 PLANIFICACIÓN INICIAL

Para expresar el tiempo utilizado en el desarrollo del proyecto, se usa la semana como forma representativa de las horas trabajadas. Si se tiene en cuenta que cada día se van a trabajar aproximadamente 4 horas, entonces la semana equivale a trabajar entre 20 y 25 horas.

Ahora se explican los puntos que se tuvieron en cuenta en un primer momento.

- **ESTUDIO PREVIO**
  Antes de empezar a programar, se tendrá que investigar sobre las principales etapas de compilación y realizar un estudio de la programación en Cosel y del módulo Com. En una semana habrá tiempo para poder llevarlo a cabo.

- **ESTUDIO LR (1) Y LALR (1)**
  Una vez obtenida la documentación necesaria para el estudio de los algoritmos, se pensó que una semana sería suficiente para tener asimilados los algoritmos LR (0), LR (1) y LALR (1).

- **DESARROLLO LR (1) Y LALR (1)**
  Desarrollo del autómata LR (1) y seguidamente el LALR (1). Se supone que este punto es uno de los más complejos del proyecto. Como consecuencia, se puso de margen cuatro semanas.

- **TABLA ANÁLISIS**

  Una vez obtenido el autómata LALR (1), se necesita una tabla para poder aplicar el análisis ascendente. Como este apartado sólo consiste en traducir el autómata a una matriz, se apostó por una semana para desarrollarlo.

- **ANALIZADOR LALR (1)**

  Finalmente, del apartado sintáctico, falta implementar el algoritmo del analizador ascendente basado en las acciones *desplazar* y *reducir*. Para realizar esta implementación, se hace necesario el uso de la tabla de análisis creada con anterioridad. En un primer momento, se pensó que en dos semanas se podría realizar. Una vez finalizado el analizador sintáctico, se tiene que probar para comprobar su buen funcionamiento.

- **ESTUDIO SEMÁNTICO**

  Una vez comprobado el buen funcionamiento del analizador LALR (1), hay que hacer un estudio sobre los diferentes atributos que puede soportar una gramática dependiente de contexto y las acciones que puede llevar a cabo. Llevará una semana realizar este estudio.

- **DESARROLLO SEMÁNTICO**

  Terminado el estudio, se ha de empezar a tratar los atributos y las acciones semánticas desde el punto de vista de un generador ascendente sintáctico. Se supone, que este apartado es otro de los más complejos del proyecto, por esta razón su margen es de cinco semanas.

- **MEMORIA Y PRESENTACIÓN**

  Aunque la memoria se va escribiendo durante todo el desarrollo del proyecto, se necesitan unas tres semanas más, para recopilar toda la información obtenida y explicar todo lo realizado de la mejor forma posible.

- **PRESENTACIÓN**

  Una vez finalizada la memoria, la presentación se puede realizar perfectamente en una semana, ya que se han de tener claros los puntos a destacar en la presentación.

2.2 PLANIFICACIÓN FINAL

Una vez finalizado el proyecto, han surgido diferentes cambios respecto a la planificación inicial.

Inicialmente, no se tuvo en cuenta la dificultad tanto en la comprensión como en el desarrollo de los algoritmos a aplicar a lo largo del proyecto. Ahora, se explican, únicamente, los apartados modificados respecto la planificación inicial.

- **ESTUDIO LR (1) Y LALR (1)**
  Se pensó que una semana sería suficiente para entender los algoritmos pero, finalmente, se incrementó en una semana más. Esto se produjo debido a las escasa información obtenida sobre los algoritmos ascendentes LR (1) y LALR (1), ya que dicha información sólo aparece en dos de los ocho libros referenciados en la bibliografía y, además, estos usan los mismos ejemplos para explicar el desarrollo de los algoritmos. Este hecho dificultó su comprensión.

- **DESARROLLO LR (1) Y LALR (1)**
  Durante el desarrollo del autómata LR (1) y LALR (1) aparecieron varios problemas que se tuvieron que resolver. Como consecuencia, el tiempo transcurrido en esta etapa es superior a lo indicado en un principio. La implementación de los autómatas se demoró un semana más.

- **TABLA ANÁLISIS**
  Para crear las tablas de análisis se tuvo un pequeño problema de optimización, que hizo incrementar el tiempo de desarrollo en tres días más.

- **ANALIZADOR LALR (1)**
  A la hora de realizar el algoritmo de análisis ascendente se tuvieron varias dificultades que hicieron retrasar de nuevo el desarrollo del proyecto. Se necesitaron unas tres semanas para su implementación.

- **DESARROLLO SEMÁNTICO**

  Es uno de los apartados en los que se han encontrado más dificultades, debido a que su implementación provocaba la modificación de una gran parte de código. Por tanto, el tiempo de elaboración se incrementó en dos semanas más de lo previsto.

- **EXÁMENES**

  Durante el tiempo que se estuvo realizando el proceso semántico se tuvo que hacer un parón a causa de los exámenes. Este periodo fue de 4 semanas.

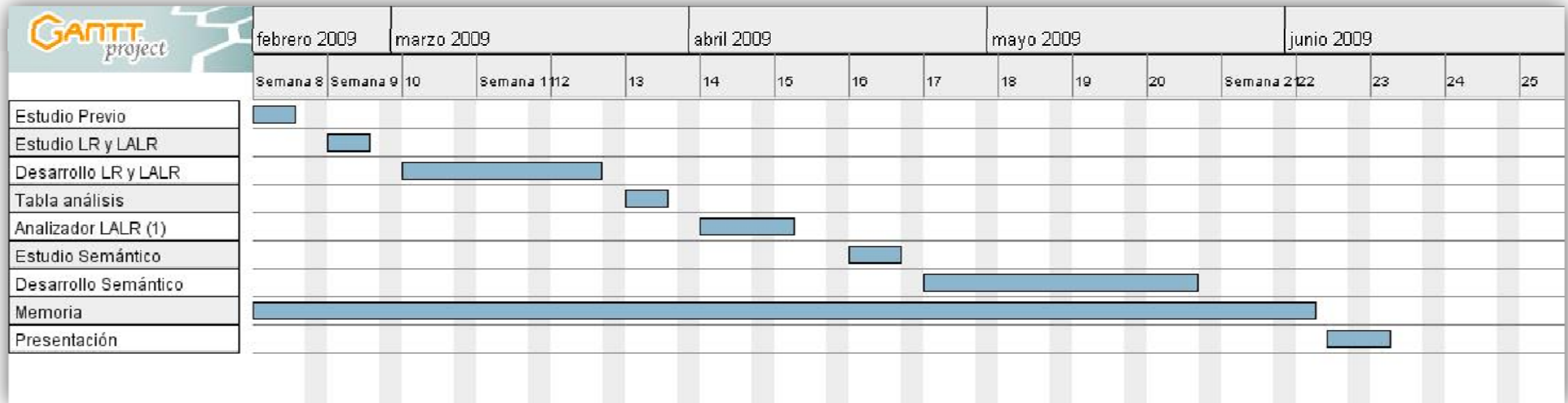
- **PERIODO DE PRUEBAS**

  Finalizados los exámenes y el desarrollo semántico, se puso a comprobar si realmente funcionaba todo. Esta sección, no se tuvo en cuenta en la planificación inicial y es una de las más importantes, ya que como suele pasar, se produjeron varios errores que retrasaron de nuevo el tiempo de desarrollo (dos semanas).

- **VACACIONES**

- **DOCUMENTACIÓN PARA LA MEMORIA**

  Este apartado no estaba incluido en la planificación inicial. Simplemente se basa en recolectar información sobre los algoritmos o explicaciones teóricas más importantes, recogida durante todo el proyecto. Más tarde, todos estos conceptos se explican y detallan en la memoria.



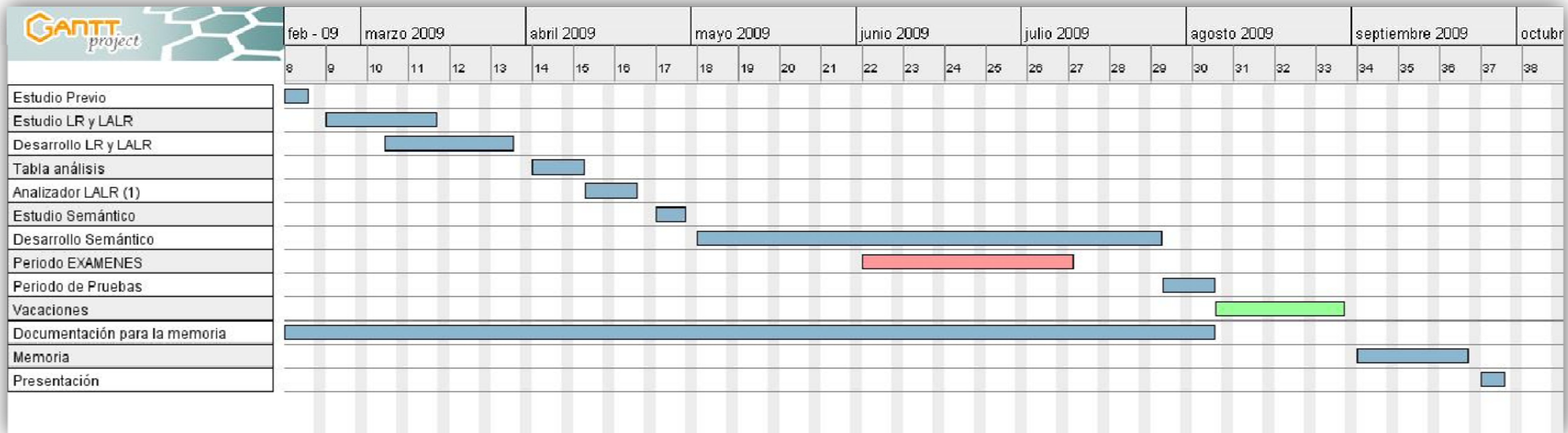


FIGURA 2.1 PLANIFICACIÓN INICIAL Y FINAL

2.3 ESTUDIO DE VIABILIDAD

A continuación, se muestra el estudio de viabilidad realizado a partir de la planificación inicial expuesta en el apartado anterior.

Primero, se ha de mirar si se tiene la información adecuada para poder generar un analizador ascendente LALR (1). Este punto, no tiene ningún riesgo, debido a que la mayor parte de la información está explicada en los libros de compiladores. También ha habido un estudio de los generadores de análisis ascendentes ya existentes, YACC y Bison, excepto en el caso de paso por parámetros ya que tienen problemas de viabilidad en los atributos heredados.

Como plataforma para hacer este analizador ascendente LALR (1) se usa el lenguaje Cosel y el módulo Com, que son de dominio público. Por tanto, para desarrollar este proyecto, se necesita un coste que es equivalente a las horas trabajadas por el ingeniero, ya que el software utilizado es gratuito.

Un riesgo que puede aparecer en la realización del proyecto es la comprensión del funcionamiento del generador Com, que usa características de Cosel, pero el director de este proyecto tiene todos los conocimientos necesarios, ya que fue él, quien desarrolló el compilador de Cosel y el módulo Com.

Una vez recopilada toda la información, se ha de desarrollar e implementar un analizador ascendente LALR (1) usando para ello el lenguaje Cosel.

Para tener un coste orientativo del presupuesto que supone realizar este proyecto, hay que sumar las horas trabajadas por el ingeniero.

Según la planificación inicial, se trabajan 20 horas cada una de las quince semanas en las que se ha llevado a cabo la realización del proyecto. Creyendo que un ingeniero cobra 30 €/hora, el coste total del proyecto asciende a 9.000€. Teniendo en cuenta la planificación final, se produce una desviación sobre el coste inicial del proyecto de 3.000 €, provocando que el presupuesto final sea de 12.000 €.

Capítulo 3

UN COMPILADOR

La traducción es un proceso que toma como entrada un texto de un programa escrito en un lenguaje y retorna como salida el texto de un programa en otro lenguaje, donde este último mantiene el significado del texto original. Un compilador traduce un lenguaje de entrada, el lenguaje fuente, a un lenguaje salida, el lenguaje objeto. Una parte importante del proceso de traducción es que el compilador avise de la presencia de errores al usuario en el programa fuente. En la Figura 3.1 se muestra un pequeño esquema de un compilador.

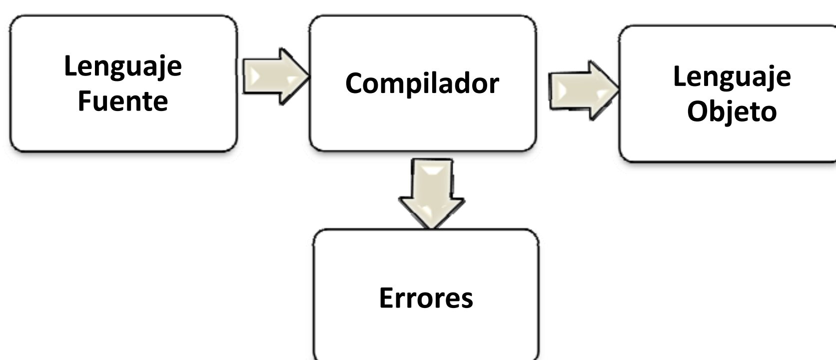


Figura 3.1 Definición de un Compilador

Un programador puede escribir un programa en un lenguaje de alto nivel, para luego compilarlo a un programa legible por la máquina.

Se pueden construir compiladores para una gran diversidad de lenguajes fuente y lenguajes máquina utilizando las mismas fases básicas. Cada una de estas fases, transforma el lenguaje fuente de una representación en otra. [4]

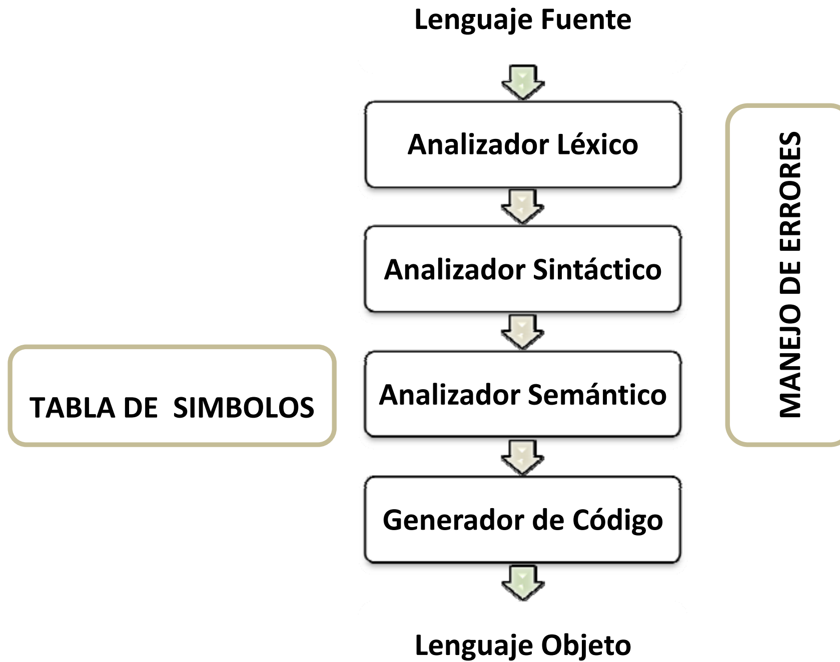


Figura 3.1 Fases de un compilador

Como se muestra en la Figura 3.2, un compilador opera en diferentes fases. Las tres primeras fases forman la mayor parte del análisis de un compilador. Aparecen dos actividades, la administración de la tabla de símbolos que se aplica en el análisis semántico y, la gestión de errores que interactúa con las etapas de análisis léxico, sintáctico y semántico.

El administrador de la tabla de símbolos es una función primordial para un compilador, ya que registra los identificadores utilizados en el programa fuente y reúne información sobre los diferentes atributos de cada identificador. [2]

Ahora se explican cada una de las fases de un compilador.

3.1 ANALIZADOR LÉXICO

El analizador léxico, también conocido como analizador morfológico (scanner, en inglés) se encarga de leer una secuencia de caracteres del programa fuente, carácter a carácter, y los

agrupa para formar unidades con significado propio. Estas unidades son los componentes léxicos (tokens, en ingés). Estos componentes pueden ser:

- **PALABRAS RESERVADAS**: *If, else, do, while, for, end, …*
- **IDENTIFICADORES**: nombres asociados a variables, funciones, tipos definidos por el usuario, etiquetas,etc.
- **OPERADORES**: *+ - * / = = < > & ( ) = ! …*
- **CONSTANTES**: constantes numéricas que representan valores enteros, reales, en punto flotante, o de caracteres que representan cadenas.

Figura 3.1.1 Analizador Léxico

Como se puede visualizar en la Figura 3.1.1 al analizador léxico se le introduce una secuencia de caracteres y este devuelve un token conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los tokens son los símbolos terminales de la gramática. Como el analizador léxico es la parte del compilador que lee el texto fuente, también puede realizar ciertas funciones además de analizar, por ejemplo, guardar el texto necesario para generar informe de errores. [5]

La forma de especificar los tokens, de modo que sean reconocidos por los analizadores léxicos, está casi siempre definida por expresiones regulares, ya que este formato resulta más cómodo para el compilador. Para más información sobre las expresiones regulares mirar la sección 6.1 del anexo.

3.2 ANALIZADOR SINTÁCTICO

La tarea del analizador sintáctico (parser, en inglés) es determinar la estructura sintáctica de un programa a partir de los tokens producidos por el analizador léxico y, de esta forma, construir el árbol sintáctico que representa esta estructura. Hay que recordar que el analizador sintáctico es la etapa más importante de la compilación (continuando el análisis iniciado por el analizador léxico), ya que comprueba que la sintaxis de la instrucción es correcta. Por este motivo, el análisis sintáctico considera como símbolos terminales las unidades sintácticas retornadas por el analizador léxico. Como se puede comprobar en la Figura 3.2.1, el analizador sintáctico recibe los tokens del analizador léxico y realiza el árbol sintáctico para, más tarde, aplicar el analizador semántico. [6]

Figura 3.2.1 Analizador Sintáctico

La estructura sintáctica a reconocer es especificada mediante una gramática libre de contexto. Las gramáticas son una herramienta esencial en el análisis sintáctico.

3.2.1 GRAMÁTICAS LIBRES DE CONTEXTO

Las gramáticas o específicamente las gramáticas libres de contexto, se usan formalmente para describir la estructura de los programas en un lenguaje de programación.

Una gramática *G*, es una tupla de cuatro componentes (T, N, S, P).

- Un conjunto de tokens, *T*, también llamados símbolos terminales, son los símbolos básicos con los que se construyen las cadenas. Los tokens constituyen los símbolos del alfabeto del lenguaje descrito por la gramática.

- Un conjunto de símbolos no terminales, *N*, designan cadenas de tokens. Cada símbolo no terminal representa un conjunto de cadenas, lo cual simplifica la definición del lenguaje generado por la gramática.
- En toda gramática existe un símbolo no terminal destacado, *S*, al que se le conoce como símbolo inicial.
- Un conjunto de producciones o reglas *P*, constan de un símbolo no terminal (llamado lado izquierdo o cabecera de la producción), una flecha y una secuencia de símbolos terminales y/o no terminales (llamado parte derecha de la producción). Estas producciones describen la forma en que se pueden combinar los símbolos terminales y no terminales para formar cadenas. A continuación se muestra la notación de una producción. [7]

```
N → (T N)*
```

Se muestra un pequeño ejemplo de una producción, donde *EXPRESION* es un símbolo no terminal y *operador* es un terminal. [6]

```
EXPRESION → EXPRESION  operador  EXPRESION
```

A partir de ahora, los símbolos terminales se expresan en letra minúscula y los no terminales en mayúsculas. Al conjunto de ellos se llama símbolos gramaticales.

3.2.2 DERIVACIÓN

La derivación consiste en tratar a cada producción como una regla de sustitución, donde el símbolo no terminal de la cabecera de una regla se sustituye por la cadena formada por el cuerpo de la producción. El árbol sintáctico se construye a partir de las derivaciones.

En el siguiente ejemplo, se observa una gramática extraída a partir del analizador léxico.

```
EXP → EXP + EXP
EXP → EXP – EXP
EXP → - EXP
EXP → (EXP)
EXP → id
```

A partir de este momento, las gramáticas se simplifican usando el operando "|" explicado en el apartado 6.1 del anexo.

EXP →EXP + EXP |EXP - EXP| - EXP | (EXP) | id

Se considera una gramática para expresiones aritméticas, donde *EXP* es un símbolo no terminal que representa una expresión. Mientras que *(*, *)*, *const* (constante) e *id* (identificador), son símbolos terminales.

La producción *EXP* → - *EXP,* indica que una expresión precedida del operador menos, también es una expresión. Esta producción se puede utilizar para crear expresiones más complejas a partir de otras más simples por el método de derivación.

Si se describe la acción anterior como *EXP* ⇒ - *EXP* se puede leer como "*EXP deriva en -EXP*". La producción *EXP*→ *(EXP),* dice que, en una cadena de símbolos gramaticales, se pueden sustituir cualquier instancia de *EXP* por *(EXP).* [8]

$$EXP \underset{EXP \rightarrow -EXP}{\Rightarrow} -EXP \underset{EXP \rightarrow (EXP)}{\Rightarrow} -(EXP) \underset{EXP \rightarrow id}{\Rightarrow} -(id)$$

3.2.3 ÁRBOLES SINTÁCTICOS

Gracias a los árboles sintácticos, se puede hacer una representación gráfica de las derivaciones. Con este método, se representa la jerarquía de la estructura sintáctica de las sentencias generadas por la gramática.

Construir un árbol sintáctico significa que se crea un árbol de nodos y que estos nodos deben ser etiquetados con símbolos gramaticales, de izquierda a derecha. Los nodos hoja son etiquetados con símbolos terminales y los nodos internos son etiquetados como no terminales. El nodo raíz (inicial) es etiquetado con el símbolo de inicio de la gramática. [8]

A continuación, se presenta una gramática, donde *EXP* y *FAC* son símbolos no terminales y *+*, *** e *id* son terminales,

EXP → EXP + EXP |FAC
FAC → FAC * FAC |id

A partir de esta gramática, se puede generar la secuencia *id*id+id* mediante los siguientes pasos de derivación. En la Figura 3.2.2, se muestra el árbol generado a partir de dicha derivación.

$$EXP \Rightarrow EXP + EXP \Rightarrow FAC + EXP \Rightarrow FAC * FAC + EXP \Rightarrow FAC * FAC + FAC \Rightarrow$$
$$id * FAC + FAC \Rightarrow id * id + FAC \Rightarrow id * id + id$$

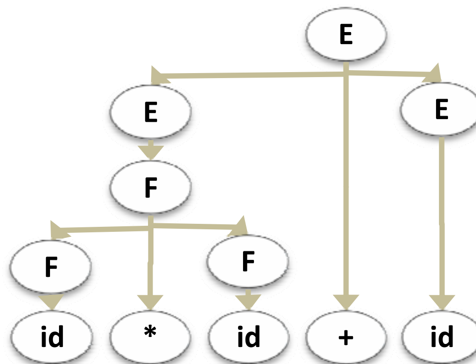


Figura 3.2.2 Árbol Sintáctico

3.2.4 MÉTODOS DE ANÁLISIS SINTÁCTICO

La mayoría de métodos de análisis se pueden dividir en dos clases: descendente y ascendente, según el orden en el que se construye el árbol.

- **ANÁLISIS DESCENDENTE** o de arriba abajo (top-down): es una técnica que intenta construir un árbol sintáctico de la cadena de entrada, comenzando por la raíz y creando los nodos a partir de ella hasta llegar a las hojas. Asimismo, se puede pensar como un método que intenta encontrar una derivada "más a la izquierda" de la cadena de entrada. [9]

- **ANÁLISIS ASCENDENTE** o de abajo arriba (bottom-up): es una técnica que pretende construir un árbol sintáctico para una determinada cadena de entrada empezando por las hojas y construyendo el árbol hasta llegar a la raíz. Si la palabra

reducción se considera el proceso inverso a la derivación, es decir, sustituir una subcadena que coincida con el lado derecho de una producción por un símbolo no terminal de la izquierda, se puede considerar que este método de análisis realiza la reducción de la cadena *c* de entrada hasta convertirla en el símbolo inicial *S* de la gramática. El resultado final coincide con una derivación por la derecha de la cadena de entrada pero escrita en orden contrario.[9]

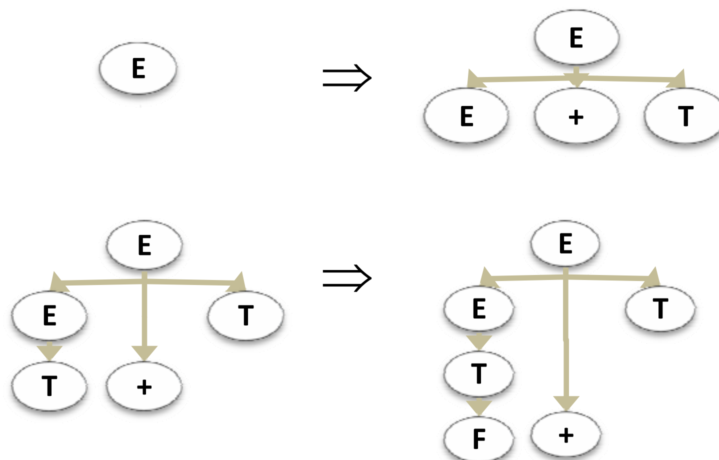
A continuación, se muestran las dos formas de aplicar la derivación en una cierta gramática cuando entra una cadena de símbolos. Los ejemplos que se realizan en cada uno de los apartados se hacen con la gramática que se muestra a continuación:

```
E  → T | E+T
T  → F | T*F
F  → id
```

- **DERIVACIÓN POR LA IZQUIERDA: (ANÁLISIS DESCENDENTE)**

  La derivación por la izquierda cosiste en reemplazar siempre el no terminal que se encuentra más a la izquierda primero. En el siguiente ejemplo se puede ver una demostración.

  E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ id+T ⇒ id+T*F ⇒ id+F*F ⇒ id+id*F ⇒ id+id*id



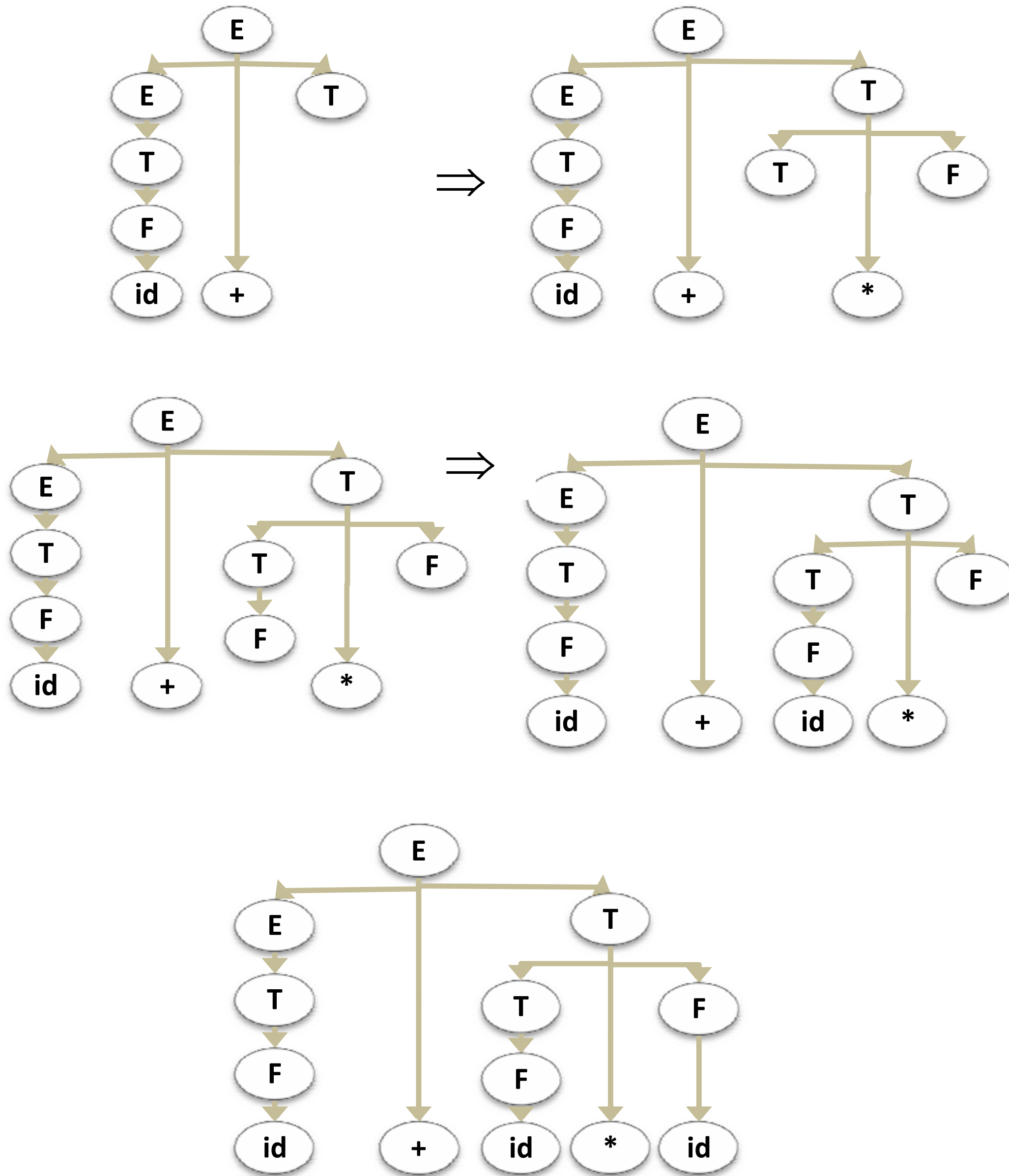
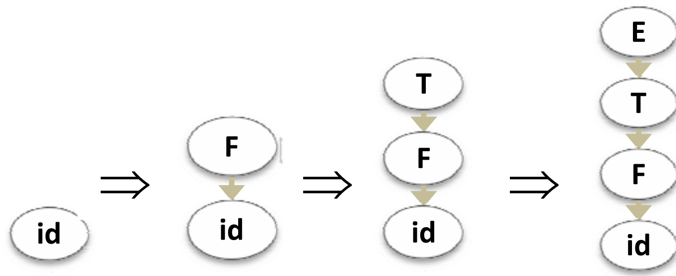


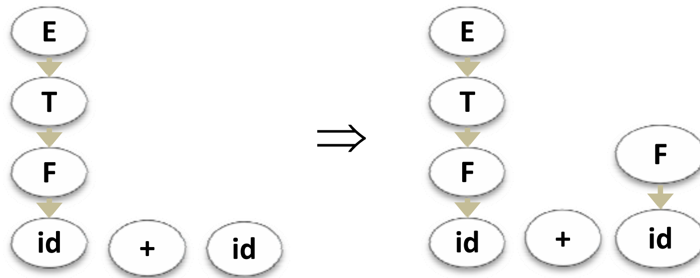
Figura 3.2.2 Análisis descendente (Derivación por la izquierda)

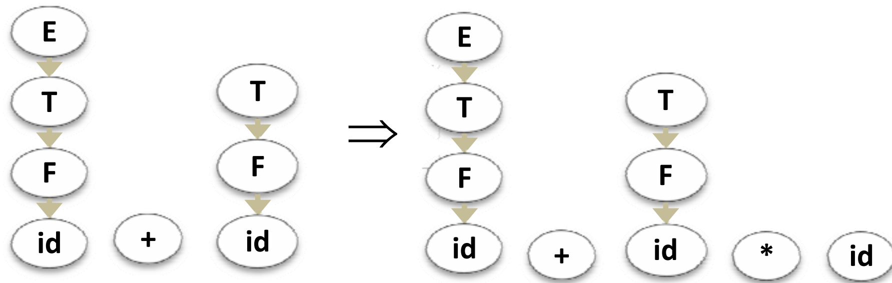
- **DERIVACIÓN POR LA DERECHA (ANÁLISIS ASCENDENTE)**

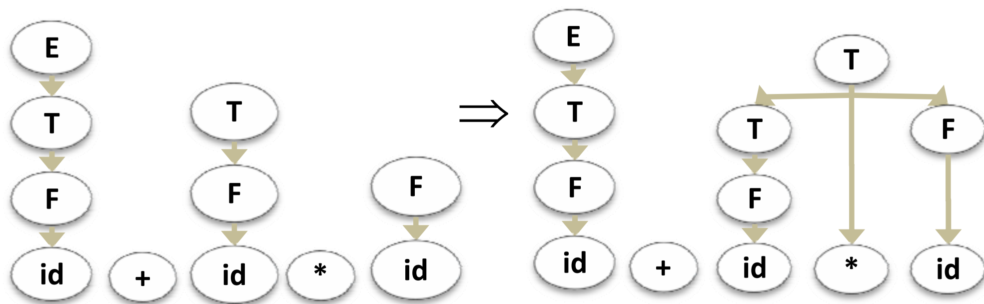
  La derivación por la derecha reside en que siempre se sustituye primero el no terminal que se encuentra más a la derecha. Y el árbol se construye desde las hojas, hasta llegar a la raíz. Se puede ver una evidencia en el siguiente ejemplo.

  id+id*id ⇒ F+id*id ⇒ T+id*id ⇒ E+id*id ⇒ E+F*id ⇒ E+T*id ⇒ E+T*F⇒ E+T ⇒ E


id
F
id
T
F
id
E
T
F
id


E
T
F
id
+
id
E
T
F
id
+
F
id


E
T
F
id
+
T
F
id
E
T
F
id
+
T
F
id
*
id


E
T
F
id
+
T
F
id
*
F
id
E
T
F
id
+
T
T
F
id
*
F
id

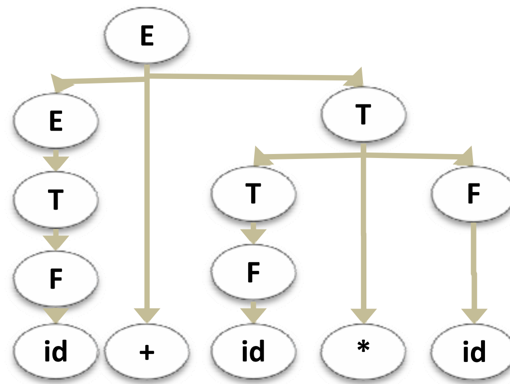


Figura 3.2.3 Análisis ascendente (Derivación por la derecha)

Como se ve en las dos Figuras anteriores, el proceso de recorrer el árbol para el análisis descendente comienza en la raíz, mientras que el ascendente empieza en las hojas, pero el resultado final es el mismo para los dos casos.

3.2.5 ANÁLISIS ASCENDENTE

Los analizadores sintácticos ascendentes, construyen y recorren el árbol sintáctico de una cadena de entrada correcta desde las hojas (símbolos terminales) hasta la raíz (axioma), en una dirección ascendente, como se ha visto en la Figura 3.2.4. Es un proceso reiterado, donde inicialmente se utiliza la cadena que se va a analizar y finaliza cuando se completa el análisis con éxito o cuando, por culpa de algún error sintáctico, no se puede continuar. En cada paso del análisis se pretende indicar que regla de la gramática se tiene que aplicar en ese punto del árbol, teniendo en cuenta el estado de este y la posición de la cadena de entrada a la que se ha llegado.

Por tanto, la parte derecha de una producción contiene una indicación, llamada punto "•", que separa la parte analizada de la que no lo está. [10]

E → • E+T

En este tipo de análisis se pueden aplicar dos acciones esenciales: reducción y desplazamiento.

- **REDUCCIÓN**

  En la parte derecha de una producción, cuando una subcadena de la parte izquierda del punto, corresponde a la parte derecha de una producción, esta se sustituye por la parte izquierda de la producción.

  Si *N* es un no terminal y deriva en el símbolo terminal o no terminal *α. β* puede ser tanto terminal como no terminal y *a* es un terminal. Se reduce una subcadena de la parte que se encuentra en la izquierda del punto *a* y se sustituye por *N*.

  $N \rightarrow \alpha$

  $\beta\ \alpha \bullet a \xrightarrow{\text{Reducción}} \beta\ N \bullet \lambda$

- **DESPLAZAMIENTO**

  Se avanza un símbolo en la cadena de entrada y, el punto en todas las reglas que siguen siendo compatibles con la subcadena de la cadena de entrada.

Se muestra un ejemplo de la utilización del punto y de las acciones Reducir y Desplazar.

$E \rightarrow \bullet\ E+T$

Inicialmente, el punto se encuentra en el origen de la parte derecha de la regla, puesto que no se ha procesado información suficiente para avanzar. Para que esta sea la regla utilizada, primero se tendrá que reducir la parte de la cadena de entrada correspondiente al símbolo no terminal *E*,

$E \rightarrow E \bullet +T$

Encontrar a continuación un símbolo terminal *+*, supone desplazar el punto una posición.

$E \rightarrow E+ \bullet\ T$

Indica que el analizador ha comprobado la parte derecha de la regla y es compatible con la parte de la cadena de entrada analizada. En este momento, se puede reducir toda la parte de la cadena de entrada asociada a *E+T* y substituirla por el símbolo no terminal *E*, que es la parte izquierda de la regla.

$E \rightarrow E+T \bullet$

Existen muchas técnicas ascendentes, al contrario de lo que ocurre en el análisis descendente, para el cual, en la práctica, sólo está disponible una técnica el LL (1). Se explica la técnica LR (0), importante desde el punto de vista teórico pero poco práctico, para después pasar a estudiar con más profundidad las técnicas de la importancia práctica LR (1) y LALR (1).

3.2.5.1 LR (K)

Se introduce una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para procesar una amplia clase de gramáticas libre de contexto, esta técnica se denomina LR (k). La abreviatura *L* indica que se realiza una lectura de izquierda a derecha de la cadena de entrada (en inglés, *from left to right*), mientras que la *R* refleja que estos analizadores construyen una derivación por la derecha de la entrada en orden inverso (en inglés, *rigth-most derivation*). Por último la *k* hace referencia al número de símbolos que se deben mirar por adelantado para tomar las decisiones de reducir o desplazar. Cuando se omite la *k*, se asume que es 1.

La diferencia que hace a los métodos LR (k) más poderosos que los LL (k) es por varia razones.

- Los métodos LR (k) disponen de la información de la entrada y los estados por lo que ha ido pasando el analizador (la cadena que ha ido reconociendo hasta ahora), mientras que los métodos LL (k) sólo disponen de la información de la entrada.
- Los analizadores LR (k) logran reconocer la gran mayoría de los lenguajes de programación que puedan ser generados mediante gramáticas libres de contexto.
- El método de funcionamiento de los LR (k) tiene la ventaja de localizar un error sintáctico casi en el mismo momento en que se produce, con lo que alcanza una gran eficiencia de tiempo de compilación frente a procedimientos menos adecuados como es el retroceso (backtracking).

El principal inconveniente del método es que supone demasiado trabajo construir manualmente un analizador sintáctico LR (k) para una gramática de un lenguaje de programación muy típico, siendo necesario utilizar una herramienta especializada para ello. Si la gramática contiene ambigüedades u otras construcciones difíciles de analizar ante un examen de izquierda a derecha de la entrada, el analizador puede localizar las reglas de producción problemáticas e informar de ellas.

Otro gran inconveniente del análisis ascendente consiste en decidir cuándo lo que parece ser la parte derecha de una producción puede ser reemplazado por su parte izquierda. Esto no es trivial, ya que pueden haber partes derechas comunes a varias producciones o producciones que derivan en final de cadena *$*.

Los analizadores LR (k) constituyen una familia en la que existen numerosas variantes. Existen varios algoritmos de análisis ascendente. El más utilizado es el algoritmo de análisis por desplazamiento-reducción. Este algoritmo está basado en una pila y una tabla de análisis, como se explica más adelante. Existen diferentes métodos para construir el autómata:

- **LR (0)**
  Es el método más fácil de implementar, pero el que tiene menos poder de reconocimiento. No utiliza la información del símbolo de adelanto para decidir la acción a realizar. Por este motivo, surgen conflictos que se mencionan más adelante.
- **SLR (1)**
  Sus siglas significan Simple LR. Es muy parecido al método anterior, pero este ya usa un símbolo de adelanto. El autómata SLR (1) tiene menos estados que el autómata LR (1), por tanto, la tabla de análisis construida con SLR (1) es mucho más compacta. Por este motivo, el conjunto de gramáticas SLR (1) es mucho más reducido que el conjunto de gramáticas LR (1).
- **LR (1)**
  Es la mejor técnica pero es muy costosa. Debido a que este autómata genera muchos estados, el conjunto de gramáticas es amplio y el tamaño de las tablas se incrementa considerablemente.
- **LALR (1)**
  Del inglés Look Ahead LR, con un símbolo de adelanto. Es una versión intermedia entre el método SLR (1) y el LR (1), ya que LALR (1) aprovecha las ventajas de ambos autómatas. El autómata es más reducido que LR (1), pero tiene un subconjunto de gramáticas más amplio que SLR (1). [9]

Ahora, se desarrollan los métodos de análisis ascendente, pero se entra más en detalle con los análisis LR (0), LR (1) y LALR (1), ya que son con los que se ha trabajado.

3.2.5.1.1 ANÁLISIS LR (0)

Los analizadores ascendentes siguen simultáneamente todos los caminos posibles, es decir, cada estado del autómata mantiene todas las reglas cuyas partes derechas son compatibles con la subcadena de entrada ya analizada. Cada una de esas reglas es hipotética, puesto que al final solo una regla es aplicada.

Se van a explicar todos los pasos que hay que tener en cuenta para crear el autómata del analizador LR (0).

- **CREAR CLAUSURA**

  A partir de una gramática y de una producción, se puede crear un estado añadiendo a este todas las producciones que puedan aparecer dependiendo de la posición del punto y de la gramática.

  Dada la siguiente gramática,

```
S’  →  •  E $
E   →  •  T
E   →  •  E + T
T   →  •  num
```

Figura 3.2.4 Gramática LR (0)

para crear el estado inicial se coge la producción inicial y el punto que se encuentra delante del primer símbolo de la cadena de entrada. Se trata de reducir toda la cadena al símbolo inicial. De esta manera, se relaciona únicamente con el símbolo inicial.

```
S’  →  •  E$
```

Un símbolo no terminal nunca se puede encontrar en una cadena de entrada, solo puede aparecer mediante una reducción. Por esta razón, cuando el analizador esta situado justo delante de un no terminal, hay que añadir todas las producciones donde su parte izquierda sea igual al este símbolo no terminal. A lo largo de la

construcción del autómata aparecen muchas configuraciones en las que hay que aplicar la reflexión anterior.

A partir de la gramática que se muestra en la Figura 3.2.5, se enseña como se crea el estado inicial *S0*. Como la primera producción tiene detrás del punto el símbolo no terminal *E*, hay que añadir al estado todas las producciones que empiecen por *E*,

```
S' → • E $
```

En este caso, las producciones que empiezan por *E*, son:

```
E  → • T
E  → • E+T
```

Por la misma razón, como *T* es un no terminal hay que añadir al estado inicial todas las reglas donde la parte izquierda coincida con el no terminal *T*.

```
T  → • num
```

Estas configuraciones tienen en común que el siguiente símbolo a analizar es un no terminal. Por eso no queda nada pendiente en el estado inicial. Como consecuencia, si se llama al estado inicial *S0*, se puede decir que:

```
S0 = {S'  → • E$
      E   → • T
      E   → • E+T
      T   → • num}
```

- **ESTADOS DE ACEPTACIÓN Y DE REDUCCIÓN**

  Un estado es de aceptación, cuando existe una producción de reducción en el estado, es decir, cuando el punto de esta se encuentra al final de la cadena y, por tanto, no precede a ningún símbolo terminal o no terminal. Se puede ver que el estado contiene el cierre de la producción *S' → E $ •*, representado en el autómata por doble círculo.

- **TRANSICIÓN ENTRE ESTADOS**

  Se van a crear todos los estados que surgen del estado inicial *S0,* dependiendo del símbolo leído en la cadena de entrada.

  Si en la cadena de entrada se encuentra el símbolo *E*, sólo la producción *(S' → • E $)* y *(E → • E + T)* siguen siendo compatibles con la entrada. En este caso, se puede desplazar un símbolo, tanto en la cadena de entrada como en estas producciones. Si el siguiente estado se llama *S1*, se puede expresar como:

  ```
  S1 = {S' → E • $,
          E  →  E • +T}
  ```

  Pero si en la cadena de entrada se encuentra el símbolo *T*, únicamente se puede desplazar un símbolo de la producción *E → • T* y de la cadena de entrada. Este estado es de aceptación, puesto que el punto se encuentra al final de la cadena. Si el siguiente estado se denomina *S2*, se puede ver como:

  ```
  S2= {E  →  T• }
  ```

  Sólo se puede producir otro estado a partir del estado inicial, si la cadena de entrada se encuentra con el símbolo terminal *num*. En este caso, se desplaza un símbolo de la producción *T → • num* y de la cadena de entrada. Este estado también es de aceptación, porque después del punto no hay más símbolos. Si este estado se llama *S3*, se expresa como:

  ```
  S3= {T  → num • }
  ```

Aplicando todos los puntos mencionados anteriormente y a partir de la gramática que se muestra en la Figura 3.2.5, se puede construir un autómata LR (0). Como se puede ver en la Figura 3.2.6.

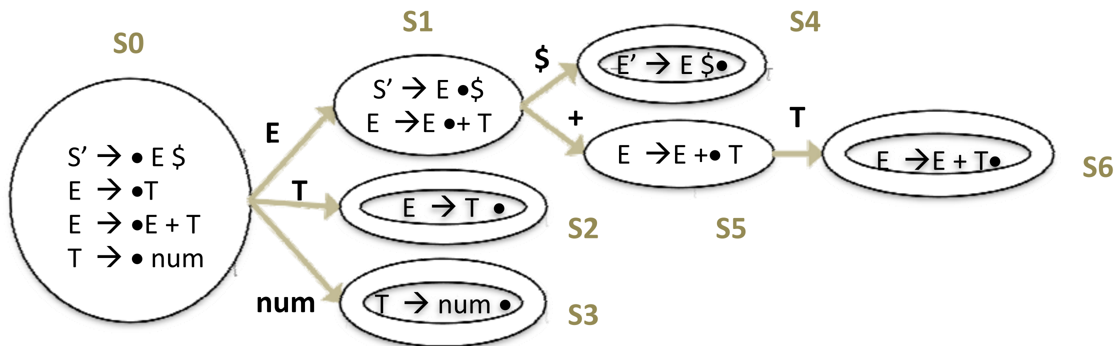


Figura 3.2.5 Autómata LR (0)

- **CONFLICTOS**

  Ocurre cuando en un estado del autómata se pueden realizar más de una acción.

  - **CONFLICTO DESPLAZAR-REDUCIR**

    Se produce cuando en un estado se consigue efectuar la acción de reducción y desplazar. Si se supone el conjunto de elementos correspondientes a un estado del autómata de análisis ascendente.

    T→ i •

    T→ i • E

    Se necesita tanto desplazar *T→ i • E* como reducir *T→ i •,* por tanto se llega a un problema.

  - **CONFLICTO REDUCIR-REDUCIR**

    Ocurre cuando en un estado aparecen dos acciones de reducir. También en el caso de tener:

    T→ i •

    V→ i •

Existe un inconveniente ya que no se puede reducir dos producciones a la vez. [10]

3.2.5.1.2 TABLA DE ANÁLISIS ASCENDENTE

Las tablas de análisis contienen información equivalente a la función de transición del autómata, que reconoce el lenguaje asociado a la gramática que se está analizando.

El número de filas coincide con el número de estados que tenga el autómata. Según la técnica aplicada se puede construir un autómata distinto con diferentes estados. Hay tantas columnas como símbolos haya en el alfabeto de la gramática. La tabla especifica la acción a realizar en cada momento. [9]

La tabla debe especificar, en función del símbolo de la cadena de entrada y del estado en el que se encuentra el autómata, cuál será el estado siguiente y las modificaciones que se deben de hacer tanto en la pila como en la cadena de entrada, según si se produce un desplazamiento o una reducción, si se ha finalizado exitosamente o si se ha detectado algún error. Para poder especificar esta información se aplicará la siguiente notación:

- Si se produce un desplazamiento se indica con ***D<e>**,* donde *<e>* representa un estado del autómata de análisis. Representa la acción de desplazar el símbolo actual e ir al estado *<e>.*
- Si se produce una reducción, se muestra con ***R<p>***, donde *<p>* identifica una regla de la gramática. Simboliza la acción de reducción de la regla *<p>.*
- Si el análisis finaliza con éxito se representa con ***Aceptar*.**
- Si no finaliza con éxito se expresa con ***Error***.

En la siguiente Figura se observa el esquema del analizador sintáctico. Donde se muestran las filas de acción y de ir_a, así como el contenido de la pila, que se explica más detalladamente a continuación. [10]

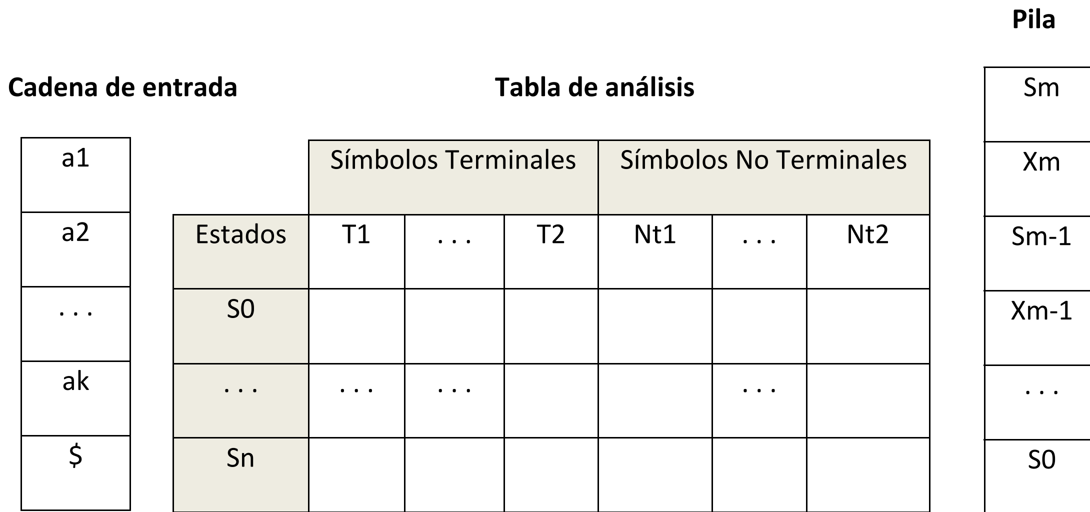


Figura 3.2.6 Estructura del Analizador Sintáctico

3.2.5.1.3 INICIO DEL ANÁLISIS

Es importante saber que el algoritmo de análisis ascendente utiliza la pila para conservar toda la información necesaria y continuar el análisis. Por esto, a excepción del estado inicial en el que se inserta el estado inicial a la pila, en el resto se introducen el estado del autómata y el símbolo examinado en ese instante.

- **DESPLAZAMIENTO D<E>**
  Cuando en la ejecución del algoritmo aparece en una casilla *D<e>* significa que, el símbolo actual de la cadena de entrada es uno de los que espera una de las reglas con la parte derecha parcialmente analizadas. Por este motivo, se puede pasar a un nuevo estado *<e>* del autómata y avanzar una posición en la cadena de entrada, de manera que el símbolo actual pase a ser el siguiente al analizado. Esto implica:

  - Introducir en la pila el símbolo de entrada.
  - Introducir en la pila el nuevo estado *<e>.*

- **REDUCCIÓN R<P>**

  Al aparecer *R<p>* simboliza que, teniendo en cuenta el símbolo de la cadena de entrada, alguna de las reglas representadas en el estado actual del autómata ha desplazado toda su parte derecha, esta puede ser substituida por el símbolo no terminal de la parte izquierda de la regla *<p>.* Esto supondrá:

  - Sacar de la pila toda la información asociada con la parte derecha de la regla *<p>.* Como al desplazar se almacena tanto el estado como el símbolo de entrada, si la longitud de la parte derecha a reducir es *N*, se tendrán que hacer *2*N* extracciones de la pila.
  - Realizar un Desplazamiento con el símbolo reducido

- **ACEPTACIÓN**

  En caso de aparecer *aceptar* en una casilla en la ejecución del algoritmo, el análisis termina y se concluye que la cadena ha finalizado correctamente.

- **ERROR**

  Si se llega a una casilla de *error* se considera que la cadena no ha finalizado con éxito y el análisis termina Para una mejor comprensión de la tabla de análisis, estas casillas se representan como casillas vacías. [10]

A continuación, se crea la tabla de análisis LR (0) a partir del autómata de la Figura 3.2.6, y se analiza la cadena de entrada *num+num$* .Mostrado en la Figura 3.2.9.

Estados	num	+	$	E	T
0	D3			D1	D2
1		D5	D4		
2					R2
3	R3				
4			Aceptar		
5					D6
6					R6

Figura 3.2.7 Tabla de análisis LR (0)

1. Lo primero que se ha de hacer, es introducir en la pila el estado 0. El primer símbolo de la cadena de entrada, es *num.* Como se puede ver en la tabla de análisis, en esa

posición hay un desplazamiento al estado 3.

2. Se añade el símbolo *num* a la pila y se extrae de la cadena de entrada. Se añade el estado *S3*. En este estado, se Reduce *(T→num).* En el estado 0, si entra una *T*, se desplaza al estado 2.

3. Se extraen de la pila dos elementos, ya que solo hay un símbolo en la parte izquierda de la producción *(T→num).* Después, se añade el símbolo *T* y el estado *S2*. Si se encuentra en el estado *S2*, se vuelve a reducir, en este caso *E→T*. Seguidamente, se debe desplazar ya que en la tabla de análisis, desde el estado *S0* con el símbolo *E*, se desplaza hacia el estado *S1*.

4. Se añaden en la pila el símbolo *E* y el estado *S1.* Ahora, entra el símbolo *+*. De esta manera, se Desplaza hacia el estado *S5.*

5. Se acumulan en la pila, el símbolo *+* y *S5*. En este paso, se vuelve a leer un símbolo de la cadena de entrada, *num*. Por tanto, hay que realizar un desplazamiento hacia *S6*.

6. Una vez añadido *num* y *S6*, se ha de reducir *T→num* y desplazar al estado donde se ha producido la reducción *S3.*

7. Se extrae de la pila los dos últimos valores introducidos, y se añade *T* y *S3*. En este paso, se reduce *E'→E+T*, y se desplaza *E'* al estado 1.

8. Como la parte derecha de la producción reducida, *E'→E+T*, tiene tres elementos, se han de extraer *3*2= 6* elementos de la pila. Después, se añade *E'* y *S1*. Ahora, entra el último elemento de la cadena de entrada *$*, que hace desplazar al estado 4.

9. Se amplía la pila, con *$* y *S4*. Por último, se reduce la producción inicial *S'→E$.*

10. El análisis ha finalizado correctamente porque en el estado 4, columna símbolo *$*, aparece la acción *Aceptar*.

Pasos	Pila	Cadena	Acciones
1	S0	num + num $	Desplaza (num, S3)
2	S3, num ,S0	+ num $	Reduce(T→num) y Desplaza(T, S2)
3	S2, T, S0	+ num $	Reduce(E→T) y Desplaza(E,S1)
4	S1, E,S0	num $	Desplaza(+, S5)
5	S5, +, S1, E,S0	$	Desplaza(num, S6)
6	S6, num, S5, +, S1, E,S0	$	Reduce(T→num) y Desplaza(T, S3)
7	S3,T, S5, +, S1, E,S0	$	Reduce(E'→E+T) y Desplaza(E',1)
8	S1,E',S0	$	Desplaza($, S4)
9	S4, $, S1,E,S0		Reduce(S'→E$)
10	S', S0		Aceptar

Figura 3.2.8 Analizador LR (0)

En este momento, se crea un autómata LR (0) con la gramática que se muestra en la Figura 3.2.10, pero en este caso se produce un conflicto en el estado 2 del autómata, donde tiene a su vez un elemento a desplazar *S→x• b* y un elemento a reducir *B→x•*. Por este motivo, se crea la técnica LR (1) que intenta resolver este conflicto restringiendo la reducción utilizando símbolos de adelanto.

S → A | x b
A → a A b | B
B → x

Figura 3.2 9 Gramática para mostrar todos los métodos

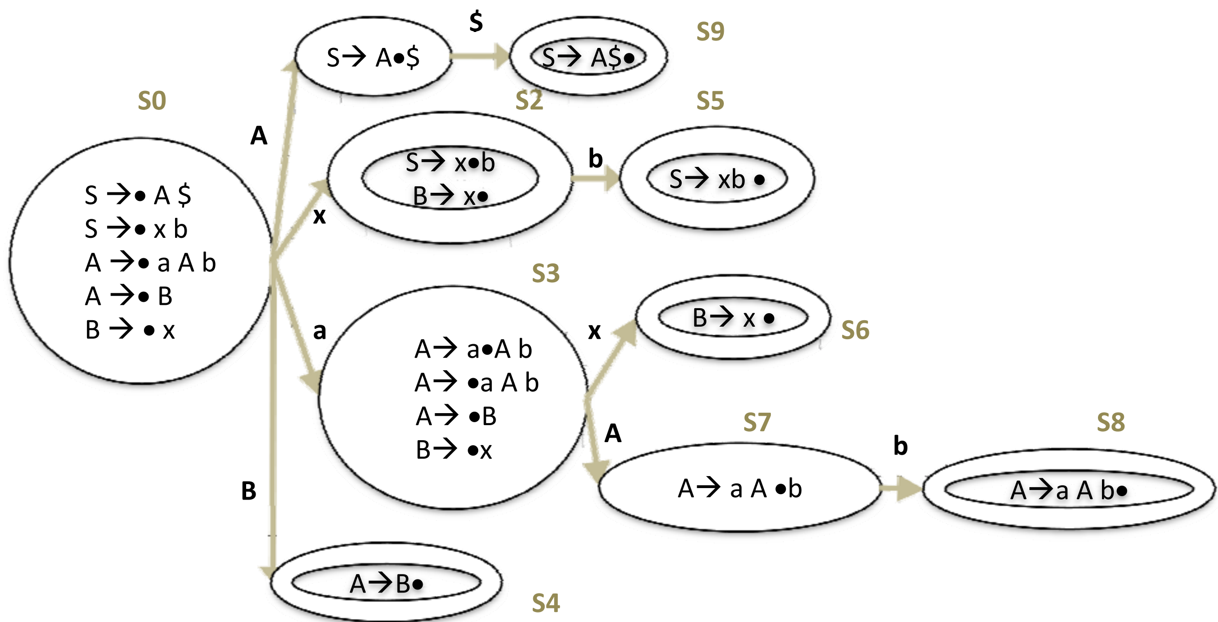


Figura 3.2.10 Autómata LR (0) con conflicto Desplazar-Reducir

3.2.5.1.4 ANÁLISIS LR (1)

El análisis sintáctico LR (1) incrementa de manera importante la potencia del análisis sintáctico LR (0) al utilizar el símbolo de adelanto en la cadena de entrada para dirigir sus acciones. Esta acción la realiza de dos maneras:

- La primera, consulta el token de entrada antes de un desplazamiento para asegurar que existe una transición apropiada.
- La segunda, utiliza un conjunto de símbolos de adelanto, para decidir si se ha de efectuar una reducción o no.

Se demuestra el análisis LR (1) a partir de la gramática que se muestra en la Figura 3.2.10. Se aplica esta gramática porque es LR (1) y LALR (1).

- Es sencillo demostrar que esta gramática no es LL (1) ya que *x* se encuentra en *PRIMEROS (B)* y, como consecuencia, en *PRIMEROS (A).* Se produce un conflicto porqué no sabe si reducir *S→A•* o avanzar a *S→x•b*.

- Como se ha explicado antes, tampoco es una gramática LR (0) porqué, como se muestra en la Figura 3.2.11, en el estado *S2* se produce un conflicto Desplazar-Reducir.

El análisis LR (1) se basa en conservar el símbolo adelantado especifico con cada elemento. De esta manera, un elemento LR (1) se escribirá así: $N \rightarrow \alpha \bullet \beta \{\delta\}$, donde δ es el conjunto de símbolos que puede seguir al símbolo no terminal β. Cuando el punto ha alcanzado el final del elemento, $N \rightarrow \alpha\ \beta \bullet \{\delta\}$, entonces el elemento se puede reducir exitosamente siempre y cuando el símbolo de adelanto en ese momento sea δ, sino es ignorado. [9]

Sólo se calculan los símbolos de adelanto para las producciones que han sido evaluadas por la clausura. Las producciones que no han sido evaluadas por la clausura mantienen el símbolo de adelanto. Para calcular los símbolos de adelanto, hay que tener en cuenta dos situaciones:

- **CONJUNTO DE PRODUCCIONES DEL ESTADO INICIAL**
  En este caso, el conjunto de símbolos de adelanto se encuentra en el estado inicial *S0*, tal y como se puede ver en la Figura 3.2.12. También, contiene solo un token, el de fin de cadena *$*, ya que este es el token que puede seguir al símbolo de inicio de la gramática.

- **CONJUNTO DE PRODUCCIONES DE CUALQUIER ESTADO**
  - Si después del no terminal *N*, hay un símbolo, por ejemplo, $P \rightarrow \alpha \bullet N\ \beta\{\delta\}$, entonces el conjunto de símbolos adelantados del nuevo elemento creado es *PRIMEROS* $\{\beta\}$, ya que esto es lo que puede continuar a cada uno de estos elementos en esta posición en concreto.
  - Pero si por ejemplo, se tiene $P \rightarrow \alpha \bullet N\ \{\delta\}$, no se puede calcular los *PRIMEROS* porqué no hay ningún símbolo después del no terminal *N*. Por esto, se coge como símbolo de adelanto $\{\delta\}$.

La definición de *PRIMEROS*, para poder incluir el conjunto de símbolos adelantados, se explica detalladamente en el apartado 6.2 del anexo.

Ahora se explica cómo se crea el estado *S0* y *S5* de autómata mostrado en la Figura 3.2.12.

En el estado *S0*, primero hay que añadir la regla de inicio *S'→• S {$},* pero como el siguiente símbolo a analizar es un no terminal, hay que agregar al estado todas las producciones donde la parte izquierda de la producción coincida con el no terminal *S*. En este caso, son *S→ • A* y *S→ • x b*. Una vez obtenidas estas producciones se calcula su símbolo de adelanto.

```
S' → • S {$}
```

Después del no terminal *S*, no hay ningún símbolo más. Por este motivo, no se pueden calcular los *PRIMEROS* y hay que añadir como símbolo de adelanto *{$}*. Por tanto, a *S → • A* y *S → • x b* se les debe añadir el símbolo de adelanto *{$}*.

```
S → • A {$}
S → • x b {$}
```

Una vez calculado sus símbolos de adelanto, hay que volver a mirar si estas producciones contienen un no terminal después del punto, como ocurre en el caso de *S→ • A {$}*, donde *A* es un no terminal. Por este motivo, se deben añadir todas las que empiecen por *A, A→ • a A b* y *A→• B*. Después de añadirlas, se calcula su símbolo de adelanto.

```
S → • A {$}
```

Ocurre como en el caso anterior, después del no terminal *A*, no hay ningún símbolo y, por tanto, no se pueden calcular los *PRIMEROS.* Por esta razón, hay que añadir como símbolo de adelanto *{$}*.

```
A → • a A b {$}
A → • B {$}
```

Finalmente, sucede igual con *A→ • B {$},* que contiene el símbolo no terminal *B*. Se añade a la clausura, *B→ • x*, que también tiene como símbolo de adelanto *{$}.*

Así es como se obtiene el estado *S0*:

```
S0= {A → a • A b {$}
     A → • a A b {$}
     A → • B {$}
     B → • x {$} }
```

El estado *S5*, se comporta diferente porque hay un símbolo adelantado *A→ a• A b {$}.*

En este caso, como después del punto hay un no terminal *A*, hay que agregar al estado todas las producciones que empiecen por *A*, que son *A→ • a A b* y *A→ • B*. Una vez añadidas, hay que calcular su símbolo de adelanto.

A→ a • A b {$}

En este caso, sí se pueden calcular los *PRIMEROS* porque, después del no terminal *A*, se encuentra *b*. Como consecuencia de producirse *PRIMEROS (b)= {b}*, el símbolo de adelanto *{b}* se añade a las producciones anteriormente cargadas.

A→ • a A b {b}
A→ • B {b}

Ahora, al analizar *A→ • B {b},* como el siguiente símbolo *B* es un no terminal, también hay que agregar todas las producciones que contengan *B*. En último lugar, se completa el estado *S5* con *B→ • x.* Al no haber ningún símbolo después del no terminal *B*, se debe de coger como símbolo de adelanto *{b}.*

Finalmente, el estado *S5* queda de la siguiente forma:

A → a • A b {$}
A → • a A b {b}
A → • B {b}
B → • x {b}

Se puede comprobar que el conflicto Desplazar-Reducir que se producía en el análisis LR (0) ha desaparecido. Ahora, el estado *S4* tiene que desplazar en *b* y reducir en *$*. En el resto de estados no se muestra ninguna alteración, por tanto, la gramática es LR (1).

La diferencia respecto a la tabla del análisis LR (0) se produce en la acción de Reducción.

- **REDUCCIÓN**

  En vez de consultar los estados finales del autómata, se utilizan los símbolos de adelanto de las producciones de reducción. Si el estado final *Sf* y su producción es *N → α• {δ1… δn},* la acción de reducción solo se añadirá en las casillas correspondientes a las columnas de los símbolos de adelanto *{δ1 … δn}.* Cuando se reduce la expresión *A→ B• {b},* en el estado *S12* de la Figura 3.2.12, se puede

comprobar que en la tabla de análisis Figura 3.2.13, en la fila 12 se reduce la regla *A →B• {b}*, en la columna donde aparece el símbolo de adelanto *b*.

- **ACEPTACIÓN**

  La producción inicial no se expresa *E' → E• $*, sino *E' → E• {$}*. El símbolo de final de cadena *$* no aparece de forma explícita, sino como símbolo de adelanto. Por tanto, la acción de aceptación se escribe en la casilla *(Si, $)*, siempre que en el estado *Si*, aparezca la producción *E' → E• {$}*. [10]

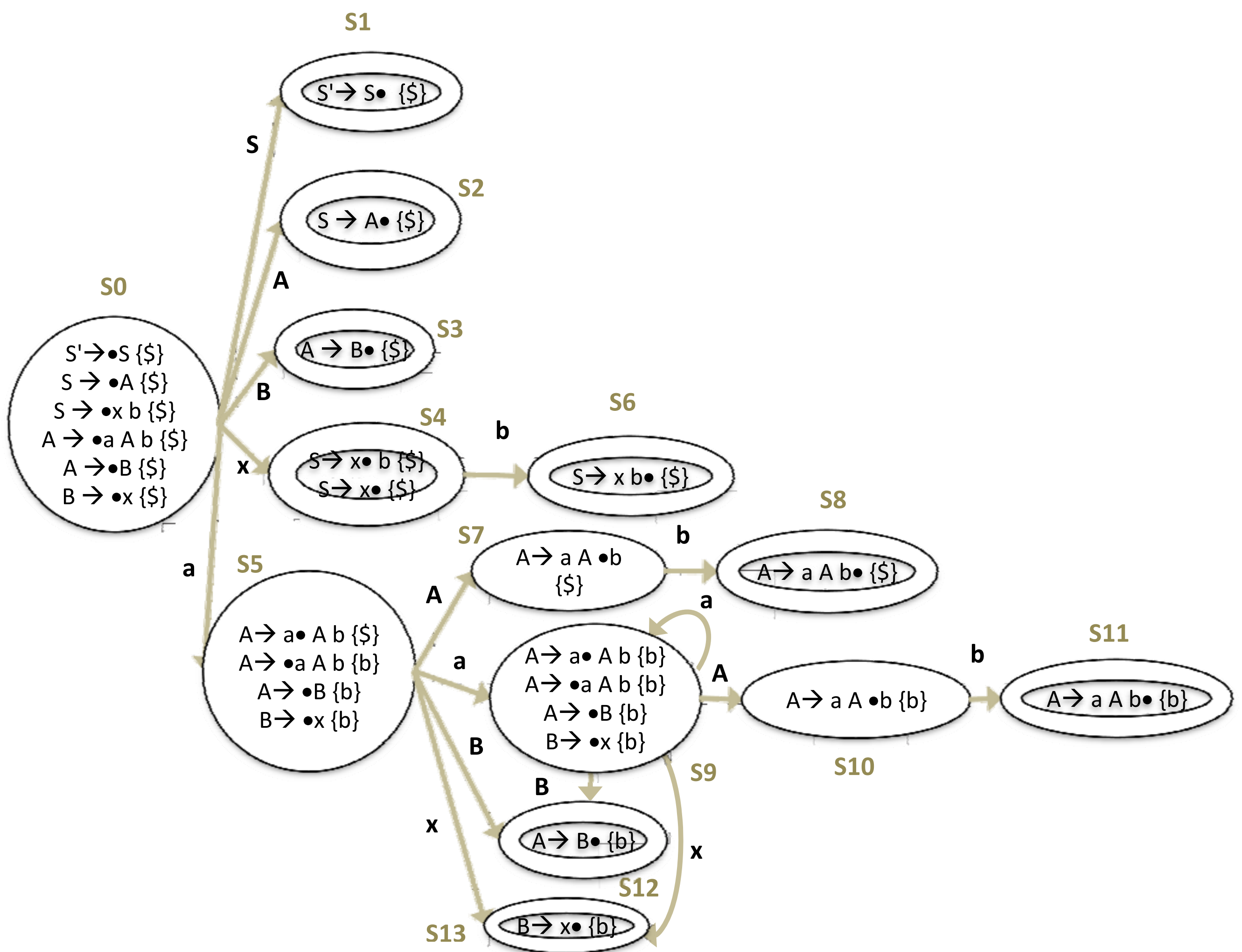


Figura 3.2.11 Autómata LR (1)

Estados	a	b	x	$	S’	S	A	B
0	D5		D4			D1	D2	D3
1				Aceptar				
2				R2				
3				R3				
4		D6		R4				
5	D9		D13				D7	D12
6				R6				
7		D8						
8				R8				
9	D9		D13				D10	D12
10		D11						
11		R11						
12		R12						
13		R13						

Figura 3.2.12 Tabla de Análisis del Autómata LR (1)

Una vez creada la tabla de análisis LR (1) mostrada en la Figura 3.2.13. Se analiza la cadena de entrada *"a x b$"*.

El análisis es muy parecido al del LR (0), lo único que se ha de tener en cuenta es el símbolo de adelanto a la hora de reducir una producción. Por ejemplo, la reducción del estado 11 y 12 se produce cuando el símbolo de adelanto de las producciones sea igual al de la cadena de entrada.

1. Como en el análisis del LR (0), se añade en un primer momento el estado 0. Si entra el símbolo *a* existe un desplazamiento a *S5*.

2. Se carga en la pila el primer símbolo de la cadena *a* y se añade el estado 5. Se vuelve a leer otro símbolo *x* y se desplaza a *S13*.

3. Se agrega en la pila *x* y el estado 13. En este momento se produce una reducción *B*→ *x*• *{b}*. Como *b* es el símbolo de adelanto hay que mirar en la fila 13, columna *b*. Después, reducir a *B* y desplazar *B* al estado 12.

4. Se extraen dos elementos de la pila porqué la parte izquierda de la producción a reducir solo contiene un elemento, *B→x•.* A continuación, se añade *B* y *S12*. En este momento se reduce *A→B•*, ya que el símbolo de adelanto coincide con *b*. Luego, se desplaza el símbolo *A* al estado *S7*.

5. Se extraen dos elementos de la pila, por la misma razón que en el paso anterior, y se añade la reducción de la regla *A* y su estado *S7*. Ahora, se lee el siguiente símbolo de la cadena de entrada *b* y se desplaza a *S8*.

6. Una vez agregados los últimos elementos, se produce una reducción *A→a A b•* y se desplaza al estado 2 para avanzar el símbolo *A*.

7. Como la reducción de antes era *A→a A b•*, ahora hay que *extraer 3*2= 6* elementos de la pila y adicionar *A* y *S2.* Se vuelve a reducir por la producción *S→A•* y desplazar el símbolo *S* al estado 1.

8. Ahora sólo falta comprobar que el símbolo de adelanto *$*, corresponde al último símbolo de la cadena *$* .Por tanto, solo queda reducir para obtener la producción inicial. Reduce*(S' →S$)*

9. La cadena ha sido aceptada

Pasos	Pila	Cadena	Acciones
1	S0	a x b $	Desplaza(a, S5)
2	S5, a, S0	x b $	Desplaza(x, S13)
3	S13, x, S5, a, S0	b $	Reduce(B→x) y Desplaza(B, S12)
4	S12 ,B, S5, a, S0	b $	Reduce(A→B) y Desplaza(A,S7)
5	S7, A, S5, a, S0	b $	Desplaza(b, S8)
6	S8, b,S7, A, S5, a, S0	$	Reduce(A→a A b) y Desplaza (A, S2)
7	S2, A, S0	$	Reduce(S→A) y Desplaza(S, S1)
8	S1, S, S0	$	Reduce(S' →S$)
9	S', S0		Aceptar

Figura 3.2.13 Analizador LR (1)

3.2.5.1.5 ANÁLISIS LAR (1)

Al examinar a fondo los estados del autómata LR (1), mostrados en la Figura 3.2.12, se observa que hay algunos elementos que son muy parecidos a otros conjuntos. Concretamente, los estados *S5* - *S9* son idénticos si se ignoran los símbolos de adelanto y lo mismo ocurre con *S3* - *S12*, *S7* - *S10* y *S8* - *S11*. Por este motivo, se intenta reducir el tamaño del autómata agrupando estos estados, y así reducir el tamaño de la tabla de análisis. Es un proceso iterativo en el que dos estados se transforman en uno solo siempre que sean equivalentes. La equivalencia de estados se basa en las siguientes condiciones:

- Sólo deben diferir en los símbolos de adelanto de las producciones. El estado que se obtiene al unir los dos estados equivalentes, contiene en cada caso la unión de los símbolos de adelanto.
- El nuevo estado agrupado debe mantener las transiciones del autómata.

El proceso se termina cuando no se puedan agrupar más estados.

Se van a ir agrupando los estados uno a uno hasta formar el autómata LALR (1), mostrado en la Figura 3.2.15.

- **ESTADOS S3 - S12**

  La asociación es posible porqué la configuración de los dos estados sólo se diferencia en los símbolos de adelanto. Las dos transiciones que llegan desde *S0* a *S3* y desde *S5* y *S9* a *S12* pueden llegar al nuevo estado *S4*. Donde:

  ```
  S3, S12 = S4 = {A→ B • {$, b}}
  ```

- **ESTADOS S7 - S10**

  Por las mismas razones que en el caso anterior se pueden juntar estos dos estados y su resultado es:

  ```
  S7, S10 = S7 = { A→ a A •b {$, b}}
  ```

- **ESTADOS S8 - S11**

  En esta situación ambos estados tienen transiciones de salida, cosa que no se había

producido en los casos anteriores. La unión es posible porqué las dos llegan al mismo estado *S7, S10=S7*. Por eso, se mantiene la transición en la generación del nuevo estado.

S8, S11 = S8 = { A→ a A b • {$, b}}

Es muy importante el orden en el que se realizan las uniones. Si se intenta unir los estados *S8* y *S11* antes que *S7* y *S10* la unión no es posible. Pero como es un proceso iterativo, esto no debe de preocupar, tarde o temprano se unificarán la pareja *S7* y *S10* y luego *S8* y *S11*. [10]

Finalmente, el autómata resultante de todas estas unificaciones es el autómata del análisis LALR (1) y se muestra en la siguiente figura:

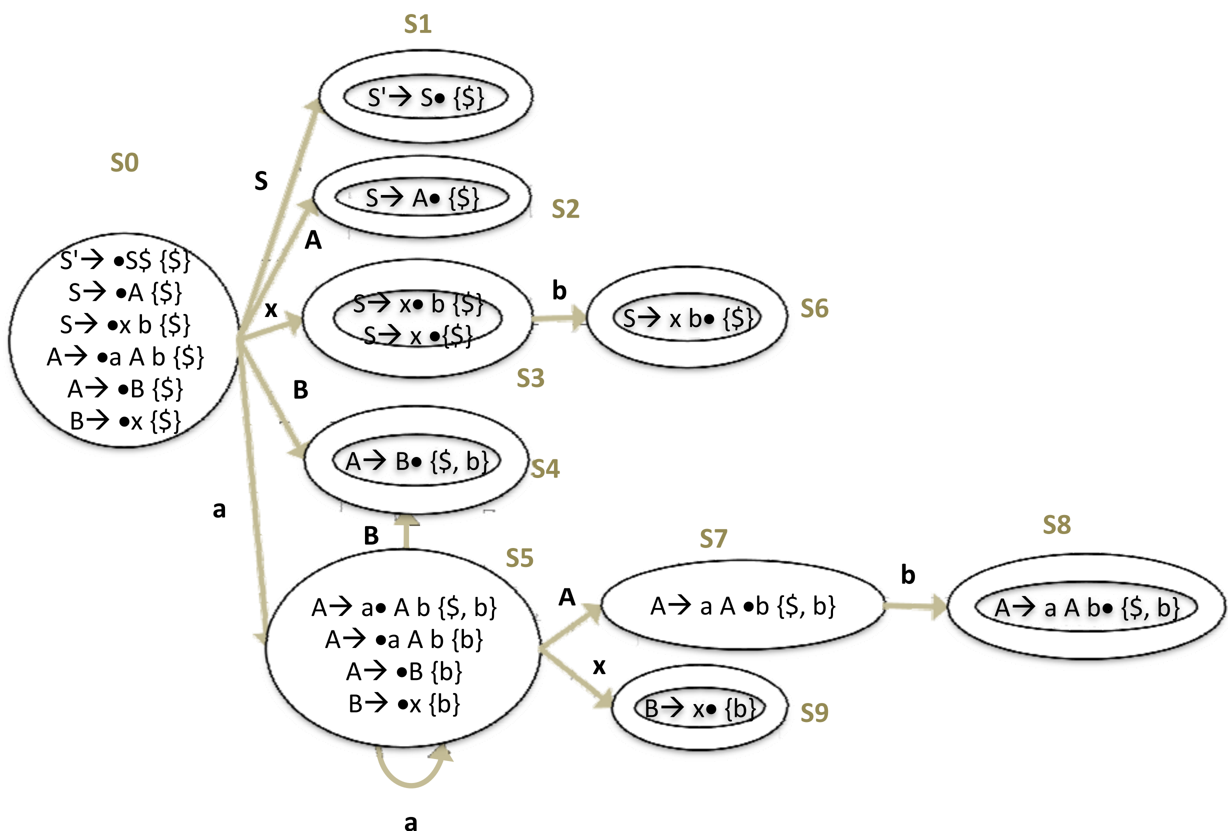


Figura 3.2.14 Autómata LALR (1)

La creación de la tabla de análisis del autómata LALR (1) se muestra en la Figura 3.2.16 y se construye exactamente igual que la tabla de LR (1).

Estados	a	b	x	$	S’	S	A	B
0	D5		D3			D1	D2	D4
1				Aceptar				
2				R2				
3		D6		R3				
4		R4		R4				
5	D5		D9				D7	D4
6				R6				
7		D8						
8		R8		R8				
9		R9						

Figura 3.2.15 Tabla de Análisis del Autómata LALR (1)

Una vez creada la tabla de análisis LALR (1) mostrada en la Figura 3.2.16. Se analiza la cadena de entrada*”a x b$”.*

Como ya se explicaron todos los pasos en el análisis LR (1), aquí no se expresarán ya que son muy parecido.

Pasos	Pila	Cadena	Acciones
1	S0	a x b $	Desplaza(a, S5)
2	S5, a, S0	x b $	Desplaza(x, S9)
3	S9, x, S5, a, S0	b $	Reduce(B→x) y Desplaza(B, S4)
4	S4 ,B, S5, a, S0	b $	Reduce(A→B) y Desplaza(A,S7)
5	S7, A, S5, a, S0	b $	Desplaza(b, S8)
6	S8, b,S7, A, S5, a, S0	$	Reduce(A→a A b) y Desplaza (A, S2)
7	S2, A, S0	$	Reduce(S→A) y Desplaza(S, S1)
8	S1, S, S0	$	Reduce(S’ →S$)
9	S’, S0		

Figura 3.2.16 Analizador LALR (1)

3.3 ANALIZADOR SEMÁNTICO

Las fases de análisis semántico y generación de código están fuertemente enlazadas y, normalmente, son coordinadas por el análisis sintáctico. El análisis semántico está dirigido por la sintaxis, ya que es el analizador sintáctico el que va invocando las rutinas al analizador semántico cuando las va necesitando. Así pues, el compilador verifica la coherencia semántica de un programa y a la vez lo traduce al lenguaje máquina o a una versión intermedia.

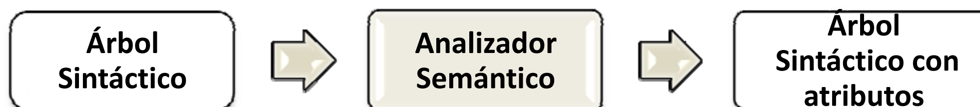


Figura 3.3.1 Analizador Semántico

Una vez realizado el análisis léxico y sintáctico, el analizador semántico recibe como entrada el árbol sintáctico del programa (véase Figura 3.3.1). El análisis semántico genera como salida, en el caso de no detectar errores, un árbol sintáctico con anotaciones semánticas o atributos. Estos atributos se emplean para comprobar que el programa es semánticamente correcto.

Para que los símbolos de una gramática pueden adquirir significado, se les asocia información (atributos) y a las producciones gramaticales se les asocian acciones semánticas, que serán código en un lenguaje de programación y cuya tarea es evaluar los atributos y tratar dicha información para llevar a cabo las tareas de traducción. Por este motivo, las gramáticas libres de contexto no son suficientes para realizar el análisis semántico. Es necesario, por tanto, definir un tipo de gramática más rica, como las gramáticas de atributos. [3]

3.3.1 GRAMÁTICA DE ATRIBUTOS

Las gramáticas libres de contexto se amplían para poder definir la semántica de un lenguaje de programación.

La gramática de atributos *GA*, es una tripleta (*G, A, R)* donde:

- G, es una gramática libre de contexto *G= (T, N, S, P). (Véase Pág. 18)*
- Cada símbolo de la gramática *S*, ya sea terminal como no terminal tiene asociado un conjunto finito de atributos *A,* y cada atributo un conjunto de valores. El valor de un atributo describe una propiedad dependiente de contexto del elemento en cuestión. Por ejemplo, la especificación *X(a,b,c),* señala que el no terminal *X*, tiene tres atributos *a*, *b*, y *c.*
- Cada producción puede tener asociada una serie de acciones *R*, que indican cómo calcular los valores de los atributos en función de los valores de los atributos de los nodos vecinos. La correspondencia entra atributos y valores es estática, es decir, un atributo de un nodo sólo puede tomar un valor determinado. [8]

Una acción semántica no solo es una asociación a un símbolo, sino que también se le puede añadir código a una regla. En los ejemplos que se van a mostrar a continuación, las acciones se van a representar entre llaves y, para acceder a los atributos, cada símbolo no terminal tiene asociado tres campos, el nombre del símbolo *nom*, el tipo del símbolo *tipo* y el valor del atributo *val*. Los símbolos terminales, tienen tres campos, el nombre del símbolo *lexnom*, el tipo del símbolo *lextipo* y el valor del atributo *lexval*, ya que se obtienen del analizador léxico. Se muestra un pequeño ejemplo, donde se escribe por pantalla el valor del atributo del símbolo *E1* y *T*.

```
E1 → E2 + T {cout.print(E1.val, T.lexval)}
```

3.3.2 TIPOS DE ATRIBUTOS

Los atributos pueden ser de dos clases, según quien les proporciona su valor.

- **ATRIBUTOS SINTETIZADOS**

  Se calculan a partir de los valores de atributos de los nodos hijos en cualquier subárbol que aparezcan. Por tanto, se trata de una información que asciende por el árbol durante su recorrido. [2]

  En la siguiente gramática todos los atributos son sintetizados:

```
E’ → E       {E’.val = E.val}
E  → E + T   {E.val = E.val + T.val}
E  → T       {E.val= T.val}
T  → T * F   {T.val= T.val * F.val}
T  → F       {T.val= F.val}
F  → num     {F.val= num.lexval}
```

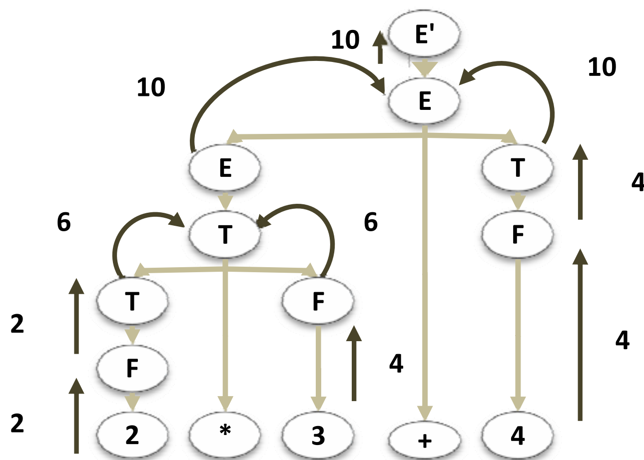


Figura 3.3.2 Atributos Sintetizados

- **ATRIBUTOS HEREDADOS**

  Se calculan a partir de los valores de atributos del nodo padre o de los nodos hermanos. Se trata de una información descendente de un lado al otro del subárbol. [2]

En el ejemplo siguiente, aparecen atributos heredados:

```
D  → L T    {L.tipo= T.tipo}
T  → entero {T.tipo =entero.tipolex}
L  → real   {T.tipo= real.tipolex}
L  → L, id  {L.tipo= L.tipo, id.tipolex=L.tipo}
L  → id     {id.tipolex=L.tipo}
```

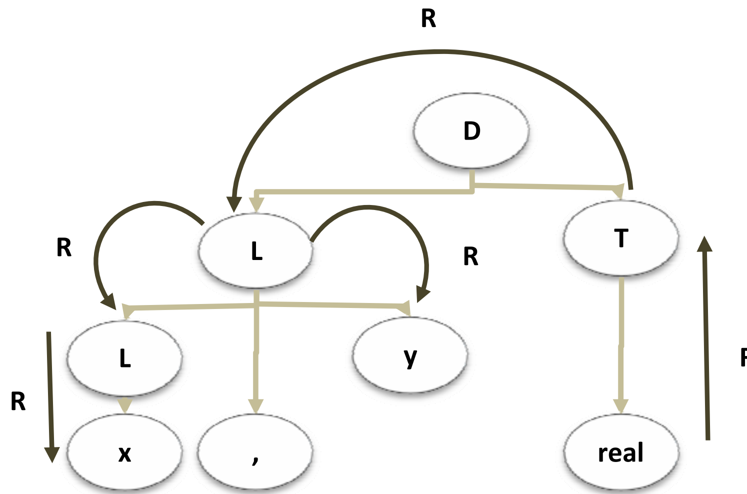


Figura 3.3.3 Atributos Heredados

Si un atributo es sintetizado o heredado debe serlo siempre en todas las producciones. Los atributos terminales son siempre sintetizados y sus valores vienen siempre dados por el analizador léxico. En cambio, los heredados sirven a menudo para expresar dependencia en el contexto en el que aparecen.

3.4 GENERACIÓN DE CÓDIGO

Es una fase del compilador que se explicará de manera general para dar una idea global sobre su funcionamiento.

La generación de código es la última fase del compilador y se encarga de convertir un programa sintáctica y semánticamente correcto en una serie de instrucciones para que puedan ser interpretadas por una máquina. Como se muestra en la Figura 3.4.1, la entrada del generador de código es un árbol sintáctico con atributos, donde la salida es un lenguaje objeto, que consiga ser desarrollado por una máquina. [4]

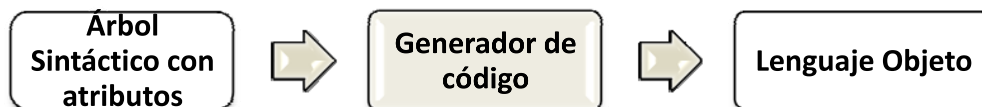


Figura 3.4.1 Generador de Código

La máquina destino puede ser un microprocesador, una máquina virtual, un lenguaje intermedio, etc. La generación de código es aplicada para crear programas de forma automática evitando que los programadores tengan que escribir código.

Capítulo 4

DESARROLLO

En este capítulo, se comenta todo el desarrollo del proyecto. Para ello se usa el lenguaje Cosel y el módulo Com, que es un generador de compiladores basado en analizadores descendentes.

Como se ha podido ver en el capítulo 3, el desarrollo del proyecto consiste principalmente en dos etapas. La primera consiste en crear un analizador ascendente partiendo del módulo Com para el análisis descendente. Esta es la parte más importante del proyecto, ya que hay que desarrollar el analizador ascendente LALR (1). La segunda etapa del desarrollo consiste en permitir al generador de compiladores Com la introducción de atributos y acciones semánticas, que serán realizadas mediante Cosel.

4.1 LÉXICO

En el apartado de análisis léxico, lo único que se ha realizado es aplicar un Scanner que ya existía previamente sobre el generador de compiladores Com de Cosel. Para de esta forma poder leer todos los símbolos de la cadena de entrada.

4.2 SINTÁCTICO

Se parte del modulo Com para el análisis descendente. El objeto gramática es muy útil, ya que permite obtener todos los símbolos terminales y no terminales; las producciones, los anulables, los primeros, etc. Todo esto es la base para crear un analizador ascendente.

Lo primero que se hace en el análisis sintáctico es pensar como almacenar cada uno de los estados del autómata LR (1). Una forma de hacerlo es añadir a la *GramaticaBNF* una estructura llamada *Estats* formada por los campos:

```
Type Estat(NumEstat, Produc, Arestes)
```

- *NumEstat*: El número correspondiente a cada estado.
- *Produc*: Las producciones que contiene cada estado.
- *Arestes*: Las transiciones que van de un estado a otro.

Pero también hay que crear una estructura *Produc* con los siguientes campos:

```
Type Produc(Valor, Punt, SimboloDeAvance, Final)
```

- *Valor*: El valor de la producción.
- *Punt*: La posición del punto dentro de la producción.
- *SímbolosDeAvance*: Los símbolos de adelanto correspondientes a la producción.
- *Final*: Es un booleano que retorna *True* si la producción ha sido finalizada y *False* si no lo ha sido.

Y *Arestes* con los correspondientes campos:

```
Type Estat(EstatAbans, Valor, EstSeg)
```

- *EstAbans*: Estado anterior al estado creado (no utilizado).
- *Valor*: Valor de la arista o símbolo con el cual se ha creado el nuevo estado.
- *EstSeg*: El nuevo estado creado.

Al objeto gramática, se le han añadido tres campos, el correspondiente a la estructura *Estats*, el campo *NumEst*, de la estructura *Estats* y el campo TablaLALR, donde se almacena la tabla de análisis ascendente LALR (1).

```
Type [public] GramaticaBNF(
    Inicial, Terminals, NoTerminals, Regles, Produccions,
    MaxNoTer, Diccionari, Anullables, Primers, Seguents,
    Estats, NumEst, TablaLALR)
```

4.2.1 AUTÓMATA LR (1)

Para introducir la explicación sobre el autómata LR (1), se va a exponer los pasos realizados por el módulo Com. Estos no son otros que pasar de BNF a Producciones, cálculo de anulables y cálculo de primeros. Todas estas funciones son necesarias, para la implementación del autómata.

Una vez obtenida toda la estructura para almacenar todos los campos del autómata, se empieza a construir el autómata.

Se inicializa la lista de estados en lista vacía y *NumEst* a cero, porqué inicialmente no existe ningún estado.

Se ha de buscar en el conjunto de símbolos no terminales, el que corresponde con el símbolo inicial de la gramática (*CodInicial*). Luego, se examina en todas las producciones de la gramática la que corresponde con *CodInicial*. Si la encuentra, se crea la nueva producción con la estructura mencionada arriba, si no provoca un error. Esto funciona correctamente en gramáticas que tengan una única producción para el símbolo inicial. Para que funcione correctamente con todas las gramáticas se tendría que generar un símbolo inicial automáticamente.

Se llama a *CrearClausura* para completar las producciones que pueda haber en un estado. Luego, se llama a *CrearEstado,* para crearlo a partir del conjunto de producciones de la clausura y, por último, se añaden las transiciones.

El código básico para crear el autómata es:

```
gl.Estats=[];
gl.NumEst=0;

var CodInicial=Found(p<-gl.diccionari,p[1]==gl.Inicial.Exp.Function)
        p[0];
var Inicio= Found (p<-gl.Produccions, p[0].cat==CodInicial)
        [Prod(p,0,[EOF],False)];

var ListaProducciones=gl.CrearClausura(Inicio);
```

```
var Estat=gl.CrearEstado(ListaProducciones);
gl.Transiciones(Estat);
gl.Estats=reverse(gl.Estats);
```

Ahora se explican cada una de las funciones de este algoritmo.

4.2.1.1 CREAR PRODUCCIÓN INICIAL

Primeramente, hay que obtener el símbolo inicial de la gramática. Se busca el símbolo inicial hasta encontrarlo, pero si no lo encuentra, se retorna *Error*. Una vez encontrado el símbolo inicial se crea su producción del siguiente modo:

```
Inicio= [Prod(p, 0, [EOF], False)]
```

Donde *p* es el valor de la producción inicial, el valor del punto se encuentra originalmente en la posición 0, el símbolo de adelanto es una lista que contiene final de fichero *[EOF]*, porque ya no hay ningún símbolo de adelanto más y como la producción no ha sido completada, el valor del booleano es *False*.

4.2.1.2 CREAR CLAUSURA

En la función *CrearClausura*, se establece el conjunto de producciones que contendrá el estado a partir de una producción.

Si el punto de la producción, está situado justo delante de un no terminal *E*, hay que añadir al conjunto de producciones todas las producciones que empiecen por el no terminal *E*. Se realiza esta operación reiteradamente hasta que no se encuentren más producciones donde el punto este situado delante de un no terminal.

Para calcular los símbolos de adelanto, hay que tener en cuenta dos situaciones:

- **CONJUNTO DE PRODUCCIONES DEL ESTADO INICIAL**
  En este caso el conjunto de símbolos de adelanto es una lista que contiene el símbolo de fin de cadena *[EOF]*, ya que es el símbolo que sigue al símbolo de inicio de la gramática.

- **CONJUNTO DE PRODUCCIONES DE CUALQUIER ESTADO**

  Cada vez que se añada una producción, hay que calcular su símbolo de adelanto. Hay dos posibilidades:

  - Cuando existe un símbolo después del no terminal *N*, por ejemplo, $P \rightarrow \alpha \bullet N \beta\{\delta\}$. Entonces, el conjunto de símbolos adelantados del nuevo elemento creado será *PRIMEROS* $\{\beta\}$, ya que esto es lo que puede continuar a cada uno de estos elementos, en particular en esta posición en concreto.
  - Pero si por ejemplo, se tiene $P \rightarrow \alpha \bullet N \{\delta\}$, como no se puede calcular los *PRIMEROS*, ya que no hay ningún elemento después del no terminal *N*, se coge como siguiente el símbolo de adelanto $\{\delta\}$.

Ahora se mostrará el algoritmo de la función Crear Clausura:

```
Fun [public] gl:GramaticaBNF.CrearClausura(ListaProducciones)=>
{
    Proc AgregarProduccion(produccion)=>
    {
        if(ListaProd.MemberP(Produccion)) return;
        ListaProd=produccion::ListaProd;
        if(!produccion.final  &&
             TypeP(NoTerminalLALR,Produccion.valor[1][Produccion.punt
             ]))
        {
             for(p<-gl.Produccions,
                    Produccion.valor[1][Produccion.punt].cat==p[0].cat
                    )
             {
                    avance=PrimersDe(produccion.valor[1].tail
                           (produccion.punt+1),
                           produccion.SimbolosDeAvance);
                    AgregarProduccion(Prod(p,0,avance,0>=p[1].length);
             }
        }
    }
    for(p<-ListaProducciones)
        AgregarProducciones(p);
    return reverse(ListaProd)
}
```

4.2.1.3 CREAR ESTADO

Una vez se obtiene el conjunto de producciones, hay que comprobar que, si al adelantar el punto de cada una de las producciones de la clausura, se ha de generar un nuevo estado o ya existe uno con las mismas producciones que el que se quiere crear.

Para poder hacer esta comprobación se mira si existe un estado con las mismas producciones que el que se va ha crear. Si ese estado ya existe, simplemente se añaden los símbolos de adelanto de la actual producción y se retorna una tupla *(Estado, False)*. Mientras que si no existe, se crea un nuevo estado, se incrementa el número del estado, se añade en el campo *Produc* el conjunto de producciones establecidas en crear clausura y se pone como *arestes* una lista vacía. Por último, se retorna una tupla con *(EstadoNuevo, True)*.

Así, cuando se crean las transiciones se sabe si hay que añadir las aristas o no, dependiendo si el segundo valor del retorno es *True* o *False*.

Ahora, se muestra el código de la función *CrearEstado*, donde *ListaProducciones* es un parámetro que representa el conjunto de producciones, una vez hecha la clausura. Con el procedimiento *AñadirAdelanto*, se añaden los símbolos de adelanto a las producciones iguales que contengan diferentes símbolos de adelanto.

```
Fun [public] gl:GramaticaBNF.CrearEstado(ListaProducciones)=>
{
    Search(e<gl.Estats,CompararProducciones(e.Produc,ListaProduccione
             s)) {
             AnadirAdelanto(ListaProducciones,e.Produc);
             return e,false;
    }
    else {
             var Estado=Estat(gl.NumEst,ListaProducciones,[]);
             gl.Estats=estado::gl.Estats;
             ++gl.NumEst;
             return Estado,true;
    }
}
```

4.2.1.4 CREAR TRANSICIONES

Una vez se obtiene el estado, solo falta añadir sus transiciones. Para esto, hay que crear un procedimiento, que cree una lista de aristas de cada uno de los estados que se vayan creando. En este apartado hay que tener en cuenta varios aspectos:

- Si en el conjunto de producciones hay más de una producción que tienen el punto delante del mismo símbolo, sólo puede aparecer una vez esta arista en la lista.
- Si existe una producción dentro del conjunto de producciones que el punto haya llegado al final, no se añade la arista.
- Una vez ya se ha construido la lista de aristas, se han de crear los nuevos estados con las producciones correspondientes. Es en este momento cuando se crea la estructura *arestes*, en la que se añade el número del estado que se había creado, el símbolo de la arista y el número del estado que se ha de crear. Por tanto, la estructura *arestes* tiene la siguiente estructura:

```
Arestes(NumEstatAnterior, ValorArista, NumEstaSiguiente)
```

- Si el estado se ha creado nuevo hay que añadir al campo *Arista* la estructura *Arestes* y, además, el conjunto de producciones que sean compatibles con la arista. Por tanto, la estructura de *Estat* es:

```
Estat(NumEstat,ConjuntoProducciones,[Arestes(NumEstatAnte
rior, ValorArista, NumEstaSiguiente)])
```

Al crear el nuevo estado, se llama a la función *CrearClausura (nuevaproduccion)*.Se crean reiteradamente estados que ya tienen como parámetro el conjunto de producciones correspondiente a ese estado. Al crear un estado, como se ha visto anteriormente, se retorna el nuevo estado y una variable booleana *nuevo*, que indica si se ha creado el estado nuevo o ya existía previamente. De esta forma, sólo en el caso de ser nuevo, se llama a la función *transiciones*, para que complete al estado añadiendo las aristas correspondientes.
A continuación, se muestra el código del procedimiento *Transiciones*, que tiene como parámetro *estado*, la primera variable de la tupla retornada por la función *CrearEstado*.

```
proc [public] gl:GramaticaBNF.Transiciones(estado)=>
{
    var lpro=estado.produc;
     while (lpro!=[]) {
          var p=lpro.head;
          var resto=if (p.final) [] else p.valor[1].tail(p.punt);

          if (resto==[]) {
               lpro=lpro.tail;
          }
          else {
                var sym=resto.head.cat;
                var lprotran=[p1|p1<-
                        lpro,p1.valor[1].tail(p1.punt)!=[],p1.valor[1].tai
                        l(p1.punt).head.cat==sym];
                lpro=lpro-lprotran;
                lprotran=[prod(p.valor,p.punt+1,p.SimbolosDeAvance,p.pun
                        t+1>=p.valor[1].length)|p<-lprotran];
                var|nuevoestado,nuevo|=gl.CrearEstado(gl.CrearClausura(l
                        protran));
                estado.Arestes=Aresta(estado.numestat,sym,nuevoestado.nu
                        mestat)::estado.Arestes;
                if (nuevo)
                        gl.transiciones(nuevoestado);
          }
     }
}
```

4.2.1.5 VER ESTADOS

El procedimiento *VerEstados*, es una forma de representar más visualmente el autómata LR (1).

En la Figura 4.2.1 se pueden ver todos los estados del autómata LR (1), de la Figura 3.2.12, sin aplicar la función V*erEstados*, es decir, cada estado es una estructura formada por el número del estado, el conjunto de las producciones y la lista de aristas, que ya se han explicado anteriormente.

```
Estat(NumEst, Prod(Valor, Punt, SimbolosDeAvance, Final),
[Aresta(EstAbans, Valor, EstSeg)])

Estat(0,[Prod((0,[1]),0,[[]],False),Prod((1,[xi,bi]),0,[[]],Fal
se),Prod((1,[2]),0,[[]],False),Prod((2,[3]),0,[[]],False),Prod(
(3,[4]),0,[[]],False),Prod((4,[xi]),0,[[]],False),Prod((3,[ai,2
,bi]),0,[[]],False)],[Aresta(0,ai,7),Aresta(0,4,6),Aresta(0,3,5
),Aresta(0,2,4),Aresta(0,xi,2),Aresta(0,1,1)])

Estat(1,[Prod((0,[1]),1,[[]],True)],[])

Estat(2,[Prod((1,[xi,bi]),1,[[]],False),Prod((4,[xi]),1,[[]],Tr
ue)],[Aresta(2,bi,3)])

Estat(3,[Prod((1,[xi,bi]),2,[[]],True)],[])

Estat(4,[Prod((1,[2]),1,[[]],True)],[])

Estat(5,[Prod((2,[3]),1,[[]],True)],[])

Estat(6,[Prod((3,[4]),1,[[]],True)],[])

Estat(7,[Prod((3,[ai,2,bi]),1,[[]],False),Prod((2,[3]),0,[bi],F
alse),Prod((3,[4]),0,[bi],False),Prod((4,[xi]),0,[bi],False),Pr
od((3,[ai,2,bi]),0,[bi],False)],[Aresta(7,ai,13),Aresta(7,xi,12
),Aresta(7,4,11),Aresta(7,3,10),Aresta(7,2,8)])

Estat(8,[Prod((3,[ai,2,bi]),2,[[]],False)],[Aresta(8,bi,9)])

Estat(9,[Prod((3,[ai,2,bi]),3,[[]],True)],[])

Estat(10,[Prod((2,[3]),1,[bi],True)],[])

Estat(11,[Prod((3,[4]),1,[bi],True)],[])

Estat(12,[Prod((4,[xi]),1,[bi],True)],[])

Estat(13,[Prod((3,[ai,2,bi]),1,[bi],False),Prod((2,[3]),0,[bi],
False),Prod((3,[4]),0,[bi],False),Prod((4,[xi]),0,[bi],False),P
rod((3,[ai,2,bi]),0,[bi],False)],[Aresta(13,ai,13),Aresta(13,xi
,12),Aresta(13,4,11),Aresta(13,3,10),Aresta(13,2,14)])
```

```
Estat(14,[Prod((3,[ai,2,bi]),2,[bi],False)],[Aresta(14,bi,15)])

Estat(15,[Prod((3,[ai,2,bi]),3,[bi],True)],[])
```

Figura 4.2.1 Estados del autómata LR (1)

En la Figura 4.2.2 se logra ver todos los estados y aristas del autómata LR (1), representados en la Figura 3.2.12 de una forma más visual, aplicando la función V*erEstados*. A continuación, se muestra la notación de cómo se representa el autómata.

```
Estado: NumEst
Produccion Izquierda → Produccion Derecha {Simbolo adelanto}
Aristas
Estado Actual – Simbolo → Estado Siguiente
```

```
ESTADO: 0
<Si>   --> · <Si_1> {<SCHAR:EOF>}
<Si_1> --> · xi bi {<SCHAR:EOF>}
<Si_1> --> · <ao> {<SCHAR:EOF>}
<ao>   --> · <ao_1> {<SCHAR:EOF>}
<ao_1> --> · <bo> {<SCHAR:EOF>}
<bo>   --> · xi {<SCHAR:EOF>}
<ao_1> --> · ai <ao> bi {<SCHAR:EOF>}
ARISTAS
0 -- ai     --> 7
0 -- <bo>   --> 6
0 -- <ao_1> --> 5
0 -- <ao>   --> 4
0 -- xi     --> 2
0 -- <Si_1> --> 1


ESTADO: 1
<Si>   --> <Si_1> ·{<SCHAR:EOF>}


ESTADO: 2
<Si_1> --> xi · bi {<SCHAR:EOF>}
<bo>   --> xi · {<SCHAR:EOF>}
ARISTAS
2 – bi --> 3

ESTADO: 3
<Si_1> --> xi bi · {<SCHAR:EOF>}

ESTADO: 4
<Si_1> --> <ao> · {<SCHAR:EOF>}

ESTADO: 5
<ao>   --> <ao_1> · {<SCHAR:EOF>}

ESTADO: 6
<ao_1> --> <bo> · {<SCHAR:EOF>}

ESTADO: 7
<ao_1> --> ai · <ao> bi {<SCHAR:EOF>}
<ao>   --> · <ao_1> {bi}
<ao_1> --> · <bo> {bi}
<bo>   --> · xi {bi}
<ao_1> --> · ai <ao> bi {bi}
```

```
ARISTAS
7 -- ai     --> 13
7 -- xi     --> 12
7 -- <bo>   --> 11
7 -- <ao_1> --> 10
7 -- <ao>   --> 8

ESTADO: 8
<ao_1> --> ai <ao> · bi {<SCHAR:EOF>}
ARISTAS
8 -- bi --> 9


ESTADO: 9
<ao_1> --> ai <ao> bi · {<SCHAR:EOF>}

ESTADO: 10
<ao>     --> <ao_1> · {bi}

ESTADO: 11
<ao_1>   --> <bo> · {bi}

ESTADO: 12
<bo>     --> xi · {bi}

ESTADO: 13
<ao_1>   --> ai · <ao> bi {bi}
<ao>     --> · <ao_1> {bi}
<ao_1>   --> · <bo> {bi}
<bo>     --> · xi {bi}
<ao_1>   --> · ai <ao> bi {bi}
ARISTAS
13 -- ai     --> 13
13 -- xi     --> 12
13 -- <bo>   --> 11
13 -- <ao_1> --> 10
13 -- <ao>   --> 14


ESTADO: 14
<ao_1>   --> ai <ao> · bi {bi}
ARISTAS
14 -- bi --> 15


ESTADO: 15
<ao_1>   --> ai <ao> bi · {bi}
```

Figura 4.2 2 Autómata LR (1) en COSEL

4.2.2 AUTÓMATA LALR (1)

Se puede observar que en el autómata LR (1) hay estados que contienen producciones muy parecidas, en las que sólo se diferencian los símbolos de adelanto. Por esta razón, se crea una función denominada *CompararProducciones*, para pasar de LR (1) a LALR (1) y así poder reducir el tamaño del autómata, agrupando estos estados tan similares.

Primero, se comprueba que las producciones de los estados sean iguales aunque contengan diferentes símbolos de adelanto. Después, se crea un nuevo estado manteniendo las mismas producciones y, en el campo de símbolos de adelanto, simplemente se añaden todos los símbolos de adelanto que había en los estados antes de agrupar. Es un proceso iterativo que termina cuando no se puedan apilar más estados.

En la siguiente Figura se puede ver, la lista de todos los estados del autómata LALR (1) representado en la Figura 3.2.15.

```
Estat(0,[Prod((0,[1]),0,[[]],False),Prod((1,[xi,bi]),0,[[]],Fal
se),Prod((1,[2]),0,[[]],False),Prod((2,[3]),0,[[]],False),Prod(
(3,[4]),0,[[]],False),Prod((4,[xi]),0,[[]],False),Prod((3,[ai,2
,bi]),0,[[]],False)],[Aresta(0,ai,7),Aresta(0,4,6),Aresta(0,3,5
),Aresta(0,2,4),Aresta(0,xi,2),Aresta(0,1,1)])

Estat(1,[Prod((0,[1]),1,[[]],True)],[])

Estat(2,[Prod((1,[xi,bi]),1,[[]],False),Prod((4,[xi]),1,[[]],Tr
ue)], Aresta(2,bi,3)])

Estat(3,[Prod((1,[xi,bi]),2,[[]],True)],[])

Estat(4,[Prod((1,[2]),1,[[]],True)],[])

Estat(5,[Prod((2,[3]),1,[bi,[]],True)],[])

Estat(6,[Prod((3,[4]),1,[bi,[]],True)],[])

Estat(7,[Prod((3,[ai,2,bi]),1,[bi,[]],False),Prod((2,[3]),0,[bi
],False),Prod((3,[4]),0,[bi],False),Prod((4,[xi]),0,[bi],False)
```

```
,Prod((3,[ai,2,bi]),0,[bi],False)],[Aresta(7,ai,7),Aresta(7,xi,
10),Aresta(7,4,6),Aresta(7,3,5),Aresta(7,2,8)])

Estat(8,[Prod((3,[ai,2,bi]),2,[[]],False)],[Aresta(8,bi,9)])

Estat(9,[Prod((3,[ai,2,bi]),3,[[]],True)],[])

Estat(10,[Prod((4,[xi]),1,[bi],True)],[])
```

Figura 4.2.3 Estados LALR (1)

Y en la siguiente Figura 4.2.4 se muestra el autómata LALR (1) después de llamar a *VerEstados*.

```
ESTADO: 0

<Si>   --> · <Si_1> { <SCHAR:EOF>}
<Si_1>--> · xi bi {<SCHAR:EOF>}
<Si_1>--> · <ao> {<SCHAR:EOF>}
<ao>   --> · <ao_1> {<SCHAR:EOF>}
<ao_1 --> · <bo> {<SCHAR:EOF>}
<bo>   --> · xi {<SCHAR:EOF>}
<ao_1> --> · ai <ao> bi {<SCHAR:EOF>}

ARISTAS

0 -- ai     --> 7
0 -- <bo>   --> 6
0 -- <ao_1> --> 5
0 -- <ao>   --> 4
0 -- xi     --> 2
0 -- <Si_1> --> 1


ESTADO: 1

<Si>    --> <Si_1> ·{<SCHAR:EOF>}


ESTADO: 2

<Si_1>  --> xi · bi {<SCHAR:EOF>}
<bo>    --> xi · {<SCHAR:EOF>}

ARISTAS

2 -- bi --> 3


ESTADO: 3

<Si_1> --> xi bi ·{<SCHAR:EOF>}

ESTADO: 5

<ao> --> <ao_1> · {bi,<SCHAR:EOF >}


ESTADO: 6

<ao_1> --> <bo> · {bi,<SCHAR:EOF >}


ESTADO: 7

<ao_1> --> ai · <ao> bi {bi,
<SCHAR:EOF>}

<ao>     --> · <ao_1> {bi}
<ao_1>   --> · <bo> {bi}
<bo>     --> · xi {bi}
<ao_1>   --> · ai <ao> bi {bi}

ARISTAS

7 – ai       --> 7
7 -- xi      --> 10
7 -- <bo>    --> 6
7 -- <ao_1> --> 5
7 -- <ao>    --> 8


ESTADO: 8
<ao_1>   --> ai <ao> · bi
{<SCHAR:EOF>}

ARISTAS

8 -- bi --> 9


ESTADO: 9
```

```
ESTADO: 4
<Si_1> --> <ao> · {<SCHAR:EOF>}
```

```
<ao_1>  --> ai <ao> bi. {<SCHAR:EOF>}

ESTADO: 10
<bo>    --> xi · {bi, <SCHAR:EOF >}
```

Figura 4.2.4 Autómata LALR (1) en COSEL

4.2.3 TABLA DE ANÁLISIS

Una vez ya está construido el autómata LALR (1), hay que crear la tabla de análisis, pero en vez de crearla como se muestra en la Figura 3.2.13, se creará un vector de listas llamado Sparce Array. Aunque sea más lento al realizar una búsqueda, no consume tanta memoria, ya que no existen posiciones vacías en la matriz. Gracias a este vector de listas, también se puede representar el autómata LR (1).

Para crear este vector, primero se crean tres nuevas estructuras, correspondientes al estado del autómata y a las acciones Reduce y Desplaza.

- **ESTADOPILA**
  Sólo muestra el estado del autómata donde se encuentra en este momento. Formado solo por un campo *Estado*.

  ```
  Type EstadoPila(Estado);
  ```

- **DESPLAZA**
  Es una estructura formada por dos campos: el primero, llamado *símbolo*, indica el símbolo que se ha de leer para poder desplazar la producción y el segundo, *estado*, que muestra el estado destino.

  ```
  Type Desplaza(simbolo,estado);
  ```

- **REDUCE**

  Es una estructura que tiene tres campos: el primero, denominado *símbolo*, que muestra el símbolo que se ha de leer para reducir la producción. El segundo, *SimbolosRegla*, que corresponde al tamaño de la parte derecha de la producción, para saber cuántos elementos se han de extraer de la pila. Y el tercero, *SimboloReduccion*, es el símbolo de la izquierda de la producción, por el cual se ha de sustituir el antiguo símbolo con su estado.

```
Type Reduce(simbolo,SimbolosRegla,SimboloReduccion);
```

Una vez se sabe como son las estructuras *Desplaza*, *Reduce* y *EstadoPila*, se crea el vector de listas.

- Si se produce un desplazamiento se añade en el vector la estructura *Desplaza*, con sus dos respectivos campos, el símbolo que se ha leído y el estado siguiente.
- Si se produce una reducción hay que agregar en el vector de análisis, la estructura *Reduce* con sus correspondientes campos, explicados anteriormente.

Una vez obtenido el vector de análisis, que se muestra en la Figura 4.2.5 a partir de las acciones producidas en el autómata LALR (1). Se ha de realizar el analizador ascendente.

```
([Desplaza(1,1),Desplaza(xi,2),Desplaza(2,4),Desplaza(3,5),Desplaza(4
,6),Desplaza(ai,7)],[Reduce(<SCHAR:EOF>,1,0)],[Reduce(<SCHAR:EOF>,1,4
),Desplaza(bi,3)],[Reduce(<SCHAR:EOF>,2,1)],[Reduce(<SCHAR:EOF>,1,1)]
,[Reduce(<SCHAR:EOF>,1,2),Reduce(bi,1,2)],[Reduce(<SCHAR:EOF>,1,3),Re
duce(bi,1,3)],[Desplaza(2,8),Desplaza(3,5),Desplaza(4,6),Desplaza(xi,
10),Desplaza(ai,7)],[Desplaza(bi,9)],[Reduce(<SCHAR:EOF>,3,3)],[Reduc
e(bi,1,4)])
```

Figura 4.2.5Vector de listas de análisis

4.2.4 ANALIZADOR LALR (1)

Si se empieza en el estado *S0* y, como entrada, se lee el primer símbolo de la cadena *Si*, hay que buscar en el vector de listas y comprobar si hay alguna acción para el símbolo *Si* en el estado *S0*. Si existe, simplemente hay que mirar cual de las dos estructuras es:

El símbolo de adelanto se obtiene a partir de la lectura de la cadena de entrada. Este símbolo corresponde al siguiente símbolo a leer. De esta manera, el analizador es capaz de saber qué dirección tomar. El símbolo de adelanto se añade en la pila en el momento de desplazar y, se ha de extraer de la pila a la hora de reducir. En el siguiente desplazamiento se vuelve a añadir.

- **DESPLAZA (SÍMBOLO, ESTADO)**
  En caso de encontrar la estructura *Desplaza (símbolo, estado)*, quiere decir que, el símbolo actual de la cadena de entrada, es uno de los que espera una de las reglas con la parte derecha parcialmente analizadas. Por este motivo, se puede pasar a un nuevo estado del autómata y avanzar una posición en la cadena de entrada, de manera que el símbolo actual pase a ser el siguiente al analizado. Esto implica:

    - Introducir en la pila el campo *estado* de la estructura desplaza.
    - Introducir en la pila el nuevo símbolo de entrada.

- **REDUCE (SIMBOLO, SIMBOLOSREGLA, SIMBOLOREDUCCION)**
  Cuando se halla la estructura *Reduce (Símbolo, SímbolosRegla, SímboloReduccion)*, quiere decir que la producción representada en el estado actual se ha de desplazar toda su parte derecha con tamaño *SímbolosRegla* y substituirla por el símbolo no terminal de la parte izquierda de la producción *(SímboloReduccion)*. Esto conlleva:

    - Sacar de la pila toda la información asociada con la parte derecha de la regla, correspondiente al tamaño *SimbolosRegla*. Como al desplazar se almacena tanto el estado como el símbolo de entrada, se tiene que extraer *2 X SimbolosRegla* elementos.
    - Introducir en la pila el símbolo no terminal de la parte izquierda de la regla (S*imboloReduccion*).

A la hora de buscar en el vector de listas, si no se encuentra ningún campo que contenga el símbolo que se está leyendo en la cadena de entrada, entonces se produce un *error*.

La cadena es aceptada cuando se reduce hasta llegar al símbolo inicial, que corresponde con el valor del no terminal 0.

```
Proc [public] gl:GramaticaBNF.AnalizadorLALR(scan)=>
{
    var SimboloAdelanto=unbound;
    Fun LeerSimbolo()=>
    {
        if (SimboloAdelanto!=unbound) {
             var tmp=SimboloAdelanto;
             SimboloAdelanto=unbound;
             return tmp;
        }
        var |cat,params...|=scan();
        TerminalLALR(cat,params)
    }
    var Pila=[LeerSimbolo(),EstadoPila(0)];
    for () {
        var SimboloActual=Pila.Head;
        var EstadoActual=Pila.Tail.Head.Estado;
        Search (p<-gl.TablaLALR[EstadoActual],p.simbolo==simboloActual.cat)
        {
            switch (p) {
                Desplaza=> {
                    Pila=EstadoPila(p.Estado)::Pila;
                    Pila=LeerSimbolo()::Pila;
                }
                Reduce=> {
                    var simboloReduccion=p.FuncionDeReduccion()(Pila);
                    for (i<- 1 .. p.SimbolosRegla*2+1) Pila=Pila.Tail;
                    Pila=simboloReduccion::Pila;
                    SimboloAdelanto=SimboloActual;
                    if (SimboloReduccion.cat==0) {
                        return;
                    }
                }
```

```
            }
        }
        else {
            throw SyntaxError("no encontrada la transición para
                ",EstadoActual, " ", simboloActual);
        }
    }
}
```

4.3 AÑADIR ACCIONES SEMÁNTICAS

En esta sección del desarrollo, una vez obtenida la estructura del análisis sintáctico, se ha de poder realizar acciones semánticas. Para ello, se han de tener claros tres conceptos.

- Primero, se ha de obtener los valores de los atributos. Cada símbolo de la gramática *S*, ya sea terminal o no terminal tiene asociado un conjunto finito de atributos. Como se muestra en este ejemplo, para poder hacer la suma, antes, se ha de saber cuales son los valores de *b, c y d.*

```
A (a)→  B(b) C(c) D(d)   {A.a= B.b + C.c + D.d}
```

- Cada producción puede tener asociada una acción semántica que indica cómo calcular los valores de los atributos. Esta acción se sitúa al final de la producción y solamente se aplica cuando la producción se ha cumplido. Como se puede ver en el caso anterior, la acción *A.a= B.b + C.c + D.d* sólo se produce cuando la regla ha sido completada.

Primero, hay que extraer los atributos de los símbolos porque, cuando se pasa de BNF a producciones, se pierden todos los parámetros y los símbolos de acción.

Para ello, hay que añadir en las estructuras de los símbolos terminales y no terminales un campo denominado *Parametres***.** De esta manera, se obtienen estos atributos como se muestra a continuación.

```
Type [public,Print=Contents] NoTerminal(Exp, Parametres)
```

```
Type [public,Print=Contents] Terminal(Cat, Parametres)
```

Al añadir el campo *Parametres* a la estructura de los símbolos terminales y no terminales, se ha modificado algunas partes del código, para que acepten la nueva estructura.

Por tanto, la cuestión es que cada no terminal de la gramática tenga asociado un atributo, y se aplique la propagación de estos atributos desde los terminales a los no terminales. Se logra dicha propagación, reduciendo una regla de producción cada vez que se que se ejecute una acción.

En el análisis sintáctico sólo se almacena en la pila una tupla con el *estado* y el *símbolo*. Mientras que ahora, para poder aplicar acciones semánticas, se le ha de añadir una tupla con el *estado, símbolo y atributo*. De esta manera, un atributo es una información asociada a un símbolo tanto terminal como no terminal. En el siguiente ejemplo, se puede ver la pila que se usa en el análisis sintáctico, y la que se va a emplear para el semántico.

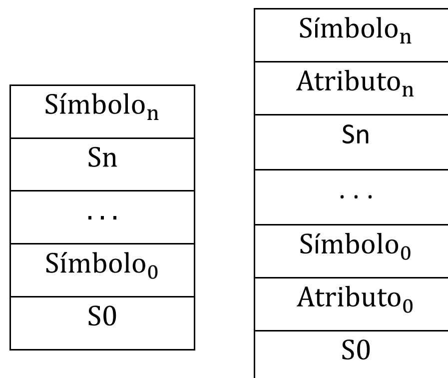


Figura4.3.1 Pila en análisis sintáctico y semántico

Para almacenar la acción de cada una de las producciones, hay que añadir un campo más a la estructura *produccions*, llamado *acción*.

```
Type Produccions(Parte Izquierda Produc, Parte derecha Produc,
Accion)
```

4.3.1 FUNCIÓN DE REDUCCIÓN

Para poder realizar lo explicado anteriormente, hay que modificar la función de crear tabla de análisis.

- Para poder ejecutar una acción semántica, hay que añadir una función anónima en el campo *SímboloReduccion* de la estructura *Reduce*, que es la que realiza la acción semántica.

```
Type Reduce(simbolo,SimbolosRegla,SimboloReduccion)
```

- La función de reducción consta de tres partes:
    1. Declaración de los atributos que se encuentran en la parte derecha de la regla.
    2. Realización de la acción semántica.
    3. Construcción del símbolo no terminal de la izquierda de la regla mediante los atributos que se habían declarado.

Se muestra un pequeño ejemplo donde se realiza la acción *de v=a*b*, para ello, como se ha explicado, primero se declaran los atributos de la parte izquierda de la regla y se les asigna el valor correspondiente almacenado en la pila. Después se realiza la acción y, finalmente, se construye el no terminal con el valor obtenido de la acción.

```
Var a = Pila.Head(6).Params[0];
Var b = Pila.Head(2).Params[0];
Var v = (a*b);
NoTerminalLALR(1,Vector(v))
```

A continuación, se muestra el código de la función *CrearTablaAccion_Ir,* donde se muestra la función de reducción.

```
Proc [public] gl:GramaticaBNF.CrearTablaAccion_Ir()=>
{
    var Matriz=new vector[gl.Estats.length];
    for (i<-Matriz:Index) Matriz[i]=[];
```

```
    for(e<-gl.Estats)
    {
        for(ar<-e.arestes)
        {
              Matriz[e.NumEstat]=Desplaza(ar.valor,ar.EstSeg)
                    ::Matriz[e.NumEstat];
        }
        for(ep<-e.Produc,ep.Final)
        {
            for(sda<-ep.SimbolosDeAvance)
            {
                // Función de reducció
                var p=ep.valor;
                var res=newapply(`vector,p[0].params.length);
                var decs= << {} >>;
                for (i<-p[0].params:Index) {
                    var param=p[0].params[i];
                    res.Arg(i)=param;
                }
                for (s<-p[1],pos<-0..;ip<-s.params:Index){
                    var dec=<< var <s.params[ip]> = Pila.Head(<(p[1].Length-
                          pos)*2>).Params[<ip>] >>;
                   decs=<< <decs>;<dec> >>;
                }
               var   f=Eval(<<   AFun(pila)=>   {   <decs>;   <ep.valor[2]>;
                   NoTerminalLALR(<ep.valor[0].cat>,<res>) } >>);
               Matriz[e.NumEstat]=Reduce(sda,ep.valor[1].length,f)::Matriz[
                   e.NumEstat];
            }
        }
    }
    gl.TablaLALR=Matriz;
}
```

Solamente se añade al autómata LALR (1) del análisis sintáctico, los atributos a cada uno de los símbolos (terminales y no terminales) y las acciones correspondientes al final de cada producción.

Seguidamente, se muestra un ejemplo con una gramática muy sencilla, donde *@var v=a+b,* representa la acción semántica de los atributos *a* y *b*:

```
Rule<ini(v)>::=<inicio(v)>
Rule <inicio(v)>::=<termino(a)> + <termino(b)> @var v=a+b;
Rule<termino(v)>::= numero #(v)
```

La notación usada para representar el autómata es la siguiente,

```
Estado: Numero del Estado
Produc Izquierda → Produc Derecha {Adelanto} @(ACCION)
Aristas
Estado Actual – Símbolo → Estado Siguiente
```

Donde acción en vez de ser representada en forma de árbol, se muestra como una simple operación, para que sea más comprensible. Por ejemplo, la acción

```
Apply(Var,[],Apply(=,v,Apply(+,a,b)
```

Se representa v=a+b

El autómata generado a partir de esta gramática es el que se indica a continuación:

```
ESTADO: 0
    <ini(v)>      --> · <Inicio(v)> {EOF} @(InstrNull)
    <Inicio(v)>   --> · <termino(a)> + <termino(b)> {EOF} @(v=a+b)
    <termino (v)> --> · Numero#(v) {+} @(InstrNull)
ARISTAS
    0 -- Numero    --> 5
    0 -- <termino> --> 2
    0 -- <Inicio>  --> 1

ESTADO: 1
    <ini(v)>   --> <Inicio(v)> · {EOF} @(InstrNull)

ESTADO: 2
    <Inicio(v)> --> <termino(a)> · + <termino(b)> {EOF} @(v=a+b)
ARISTAS
    2 -- +  --> 3

ESTADO: 3
```

```
    <Inicio(v)> --> <termino(a)> + · <termino(b)> {EOF}@(v=a+b)
    <termino (v)> --> · Numero#(v) {<EOF>} @(InstrNull)
```

ARISTAS

```
    3 -- Numero --> 5
    3 -- <termino> --> 4
```

ESTADO: 4

```
    <Inicio(v)> --> <termino(a)> + <termino(b)> · {EOF} @(v=a+b)
```

ESTADO: 5

```
  <termino (v)> --> Numero#(v) · {+, EOF} @(InstrNull)
```

Figura 4.3.2 Autómata LALR (1) con atributos

Se puede ver en la Figura 4.3.2, que la acción se añade en todos los estados donde aparece la producción

```
Rule <inicio(v)>::=<termino(a)> + <termino(b)>
```

Pero esta, sólo se aplica cuando la producción ha finalizado, es decir, en el estado 4. Ahora, se ve cómo actúa el analizador LALR (1), cuando recibe como entrada *"10 + 5"*.

El resultado de Cosel se muestra de esta forma:

```
Pila   -->[TerminalLALR(Numero,Vector(10)),EstadoPila(0)]
Accion -->Desplaza(Numero,5)
```

Para poder ver los pasos de una forma más clara se simplifica de la siguiente forma,

```
Pila   -->[Numero(10),S0]
Accion -->Desplaza (Numero,S5)
```

Ahora, se muestra el resultado de la ejecución de forma más visual:

```
Pila   --> [Numero(10),S0]
Accion --> Desplaza (Numero, S5)
```

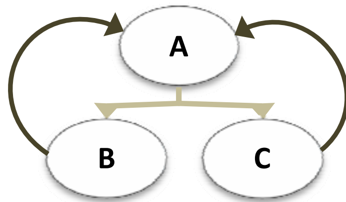
```
Pila   --> [+(), S5, Numero(10),S0]
Accion --> Reduce (+,1, FuncionAnonima)

Pila   --> [termino(10), S0]
Accion --> Desplaza (termino, S2)

Pila   --> [+(), S2,Termino(10),S0]
Accion --> Desplaza (+, S3)

Pila   --> [Numero(5),S3,+(),S2,Termino(10), S0]
Accion --> Desplaza (Numero, S5)

Pila   --> [EOF(),S5, Numero(5),S3, +(), S2, Termino(10),S0]
Accion --> Reduce (EOF, 1, FuncionAnonima)

Pila   --> [Termino(5),S3,+(), S2,Termino(10),S0]
Accion --> Desplaza (Termino, S4)

Pila   --> [EOF(),S4,Termino(5), S3,+(),S2,Termino(10),S0]
Accion --> Reduce (EOF,3, FuncionAnonima)

Pila   --> [Inicio (15),S0]
Accion --> Desplaza (Inicio,1)

Pila   --> [EOF(),S1,Inicio (15),S0]
Accion --> Reduce (EOF,1, FuncionAnonima)

Aceptado [ini(15), S0]
```

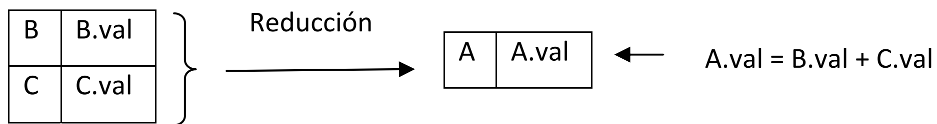
Figura 4.3 3 Análisis LALR (1) con atributos

4.3.2 ATRIBUTOS HEREDADOS

En este proyecto sólo se aplican **atributos sintetizados**. El nodo padre, puede obtener el valor de un atributo cuando se efectúa la reducción. Por ejemplo, para que *A* obtenga el valor de la suma, se ha de reducir la producción y, en este momento, se aplica la acción semántica *A.val= B.val + C.val.*

```
A  → B C {A.val= B.val + C.val }
```

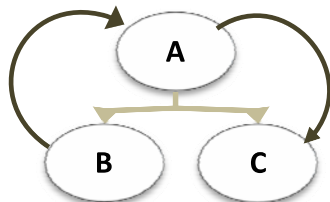




En el caso de **atributos heredados**, calculados a partir del nodo padre o de los nodos hermanos, el valor de estos atributos no se encuentra en la pila hasta que no se reduce la producción.

En el ejemplo siguiente,

```
A → B C {C.val= A.val + B.val}
```



No se puede obtener el valor de *C* hasta que no se reduzca la producción, *A*→ *B C*, pero la producción no se puede reducir porque falta el valor de *A.*

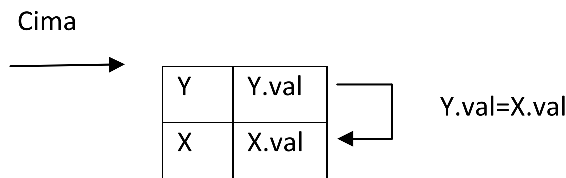
YACC resuelve el problema, mediante desplazamientos negativos en la pila. Pueden aparecer varios casos:

- Acceso, como atributo heredado, al atributo (sintetizado) de un hermano por la izquierda.
  Un símbolo puede utilizar el atributo sintetizado de un hermano suyo porque ya se encuentra en la pila. [2]

  En el siguiente ejemplo:

  ```
  A→X Y {Y.val=X.val}
  ```

  Donde *A.val* y *X.val* son atributos sintetizados y *Y.val* es heredado. Se puede acceder al valor de *X.val* porque todavía se encuentra en la pila.

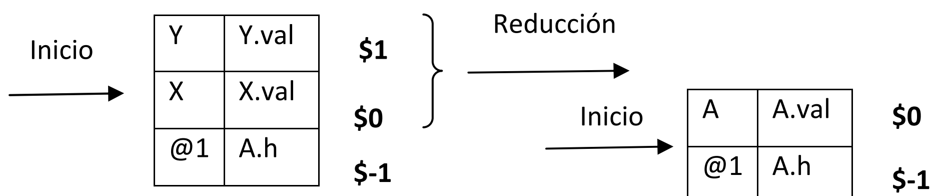


- Acceso, al atributo heredado del nodo padre. Si se supone *A* →*X Y*.

  Los atributos heredados de *A* no pueden estar almacenados en *A*, ya que no se reserva espacio para *A* hasta que no se reduce, quitando *X* e *Y* de la pila.

  La solución es que los atributos heredados no se almacenan con el símbolo sino con otro símbolo anterior introducido. Por ejemplo,

  ```
  A→X Y {X.val=A.val}
  ```



De esta forma, los atributos *X* e *Y* pueden heredar los atributos de *A* accediendo a desplazamientos negativos, respecto a *Inicio*.

Capítulo 5

CONCLUSIONES

Gracias a este proyecto se ha aprendido a crear un generador de compilador basado en analizadores ascendentes. Todo este proceso ha servido para conocer y entender cada una de las fases de un compilador, en especial al análisis sintáctico LALR (1).

Hay que destacar, todos los conocimientos adquiridos con la utilización de Cosel. También esta experiencia ha servido para poner a prueba los conocimientos adquiridos durante toda la carrera.

Finalmente los objetivos logrados han sido:

SINTÁCTICO: Se ha conseguido ampliar el modulo Com para examinar un analizador ascendente LALR (1). Para ello se ha tenido que crear una autómata LALR (1), crear una tabla de análisis, que contiene todas las acciones que puede generar la gramática y, finalmente, el analizador LALR (1), que comprueba si se puede crear el analizador a partir de la gramática ascendente LALR (1) y si una cadena de entrada es aceptada por el analizador.

SEMÁNTICO: Se ha logrado añadir a la gramática de entrada, atributos y acciones a las producciones. Por tanto, el analizador ascendente es capaz de declarar estos atributos, para poder aplicar estas acciones. Pero sólo es capaz de añadir atributos sintetizados, no heredados.

Como líneas futuras se puede considerar la posibilidad de añadir a la gramática atributos heredados, para poder tratarlos y finalizar la generación del compilador ascendente.

Capítulo 6

Anexo

6.1 Expresiones Regulares

Una expresión regular es una fórmula que describe un posible conjunto infinito de cadenas, que se ajustan a un determinado patrón. Donde λ, representa una expresión regular que representa la secuencia vacía.

La expresión regular más básica es un patrón que selecciona solo un carácter. Por ejemplo, el patrón *a*, que selecciona el carácter *a*. [9]

Existen tres patrones básicos:

- **Concatenación**

  Se produce una concatenación entre dos caracteres *a* y *b*. *ab* donde $a \in A$ y $b \in B$.

- **Unión**

  Este operando / separa alternativas. Por ejemplo, *ab / cd* selecciona *ab* o *bc*.

- **Repetición**

  b?: indica un carácter *b* opcional

  b:* indica una posible secuencia vacía de caracteres *b*. Por ejemplo, *{λ, b, bb, bbb, . . . }*

  b+: indica una posible secuencia da caracteres *b*. Por ejemplo, *{b, bb, bbb, . . . }*

6.2 CÁLCULO DE ANULABLES

Para toda gramática es posible definir anulables como el conjunto de símbolos no terminales, *A*, que puede derivar, en uno o más pasos, en la palabra vacía *(λ).*

Se aplica el siguiente algoritmo para encontrar el conjunto de símbolos anulables de la gramática.

Anulables_{S0} = {A Є No Terminal| A→λ}
Repetir
    $\text{Anulables}_{S\,n+1}$ = $\text{Anulables}_{S\,n}$ U {A Є No Terminal| A → $\alpha_1, \alpha_2, \dots, \alpha_m$
        ∀ i Є 1 .. m α_i Є $\text{Anulables}_{S\,n}$}
Hasta que $\text{Anulables}_{S\,n}$ == $\text{Anulables}_{S\,n+1}$

Se muestra un pequeño ejemplo, para una mejor comprensión, donde los símbolos en mayúsculas son no terminales y los que están en minúscula son terminales. [9]

A partir de la siguiente gramática:

I→ D I A	B→ g	D→ C A
A→ f	B→ λ	I→ h
A→ λ	C→ A B	

Se puede ver que en el primer paso los símbolos *A* y *B* derivan en *λ*.

En el segundo paso, se comprueba que *C* deriva en *A B*, por tanto *C* deriva también en *λ*.

Como el símbolo *D* contiene *C* y *A* que ya son anulables, *D* también lo es. En el siguiente paso, se comprueba que no hay ningún símbolo que derive ni de *A*, *B*, *C* ni *D*. Por tanto el conjunto se mantiene igual que el anterior, y ya se ha finalizado. [7]

Anulables_0 = {A, B}
Anulables_1 = {A, B, C}
Anulables_2 = {A, B, C, D}
Anulables_3 = {A, B, C, D}

6.3 CÁLCULO DE PRIMEROS

Si α es una cadena de símbolos gramaticales, se considera *PRIMEROS (α)* como el conjunto de terminales que inician las cadenas derivadas de *α*.

Para calcular *PRIMEROS (α)* basta con seguir las siguientes reglas para cada regla de producción de la gramática:

1. *PRIMEROS (λ)={ λ }*
2. *PRIMEROS (a β)= {a}*, donde a pertenece a un símbolo terminal y *β* es una subcadena de símbolos, tanto terminales como no terminales.
3. *PRIMEROS (A β),* donde *A* es un no terminal. Hay que seguir el siguiente algoritmo:

```
Si A ∈ Anulable entonces P0(A)={λ}
SiNo P0(A)= {∅}

Repetir Para cada Producción tal que: A→α de G
   Pn+1(A)= Pn(A) U PRIMEROS (α)
Hasta que A ∈ no terminal: Pn(A)== Pn+1(A)
```

Se mostrará un ejemplo para ver el cálculo detallado de los *PRIMEROS*, para la siguiente gramática que describe operaciones aritméticas:

```
I→ T B                          F→x
T→ F C                          C→ * F C | / F C | λ
B→ + T B | - T B | λ
```

Como los símbolos *B* y *C* son anulables en el primer paso se añade $P_0(B)= P_0(C)=$ *{λ},* mientras que el resto es conjunto vacio. El resto de pasos se demuestra en la siguiente tabla. [7]

Pasos	P (I)	P (T)	P (B)	P (F)	P(C)
0	Ø	Ø	{ λ}	Ø	{ λ}
1	Ø	Ø	{ λ, +, -}	{x}	{ λ, *, /}
2	Ø	{x}	{ λ, +, -}	{x}	{ λ, *, /}
3	{x}	{x}	{ λ, +, -}	{x}	{ λ, *, /}
4	{x}	{x}	{ λ, +, -}	{x}	{ λ, *, /}

Por tanto los *PRIMEROS* son:

PRIMEROS (I)= {x}

PRIMEROS (T)= {x}

PRIMEROS (B)= {λ, +, -}

PRIMEROS (F)= {x}

PRIMEROS (C)= {λ, *, /}

6.4 EJEMPLOS DEL ANALIZADOR ASCENDENTE LALR (1)

1. Análisis LALR (1) a partir de la gramática que se muestra en la Figura 3.2.10 con cadena de entrada *a x b*.

```
Pila   --> [ai(),S0]
Accion --> Desplaza (ai,S8)

Pila   --> [xi(),S8, ai(),S0]
Accion --> Desplaza (xi, S11)

Pila   --> [bi(), S11, xi(),S8, ai(),S0]
Accion --> Reduce (bi,1, FuncionAnonima)

Pila   --> [bo(),S8, ai(),S0]
Accion --> Desplaza (bo,S7)

Pila   --> [bi(), S7, bo(),S8, ai(),S0]
Accion --> Reduce (bi,1, FuncionAnonima)

Pila   --> [ao1(),S8, ai(),S0]
Accion --> Desplaza (ao1,S6)

Pila   --> [bi(),S6, ao1(),S8, ai(),S0]
```

```
Accion --> Reduce (bi,1, FuncionAnonima)

Pila   --> [ao(),S8, ai(),S0]
Accion --> Desplaza (ao,S9)

Pila   --> [bi(),S9, ao(),S8, ai(),S0)]
Accion --> Desplaza (bi,S10)

Pila   --> [EOF(),S10, bi(),S9, ao(),S8, ai(),S0]
Accion --> Reduce (EOF,3, FuncionAnonima)

Pila   --> [ao1(),S0]
Accion --> Desplaza (ao1,S6)

Pila   --> [EOF(),S6, ao1(),S0]
Accion --> Reduce (EOF,1, FuncionAnonima)

Pila   --> [ao(),S0]
Accion --> Desplaza (ao,S5)

Pila   --> [EOF (), S5, ao(),S0)]
Accion --> Reduce (EOF,1, FuncionAnonima)

Pila   --> [Si1(),S0]
Accion --> Desplaza (Si1,S2)

Pila   --> [EOF(),S2, Si1 (),S0]
Accion --> Reduce (EOF,1, FuncionAnonima)

Pila   -->[Si(),S0]
Accion -->Desplaza (Si,S1)

Pila   --> [EOF (), S1, Si(),S0]
Accion --> Reduce (EOF,1, FuncionAnonima)

Aceptado [ini(),S0]
```

2. Análisis LALR (1) a partir de la gramática que se muestra en la Figura 3.2.10 con cadena de entrada *a x*. En este caso, la cadena de entrada es rechazada por el analizador LALR (1), debido a que no es aceptada por la gramática ascendente.

```
Pila   -->[ai(),S0]
Accion --> Desplaza(ai,S8)

Pila   --> [xi(),S8, ai(),S0]
Accion --> Desplaza (xi,S11)

Pila   --> [EOF(),S11, xi(),S8, ai(),S0]
Error in line 27 From eja.csl
Loaded From init.csl
Loaded From Inicialization init
ai xi
     ^ <SyntaxError:0342625C>
EXCEPTION <SyntaxError:0342625C>: no encontrada la transición para
11 TerminalLALR(<SCHAR:EOF>,Vector())
```

3. A partir de la gramática que se muestra a continuación, se representa el análisis de la cadena 10+5+3.

```
ini(v) → E (v)
E (v)  → T (v)
E (v)  → E(a)> + T (b) @ var v=a+b;
T (v)  → numero#(v)
```

```
Pila   --> [Numero(10),S0]
Accion --> Desplaza(Numero,S4)

Pila   --> [+(),S4, Numero(10),S0]
Accion --> Reduce(+,1, FuncionAnonima)

Pila   --> [T(10),S0]
Accion --> Desplaza (T,S5)
```

```
Pila   --> [+(),S5, T(10),S0]
Accion --> Reduce(+,1, FuncionAnonima)

Pila   --> [E(10)),S0]
Accion --> Desplaza(E,S1)

Pila   --> [+(),S1,E(10), S0]
Accion --> Desplaza(+,S2)

Pila   --> [Numero (5),S2, +(),S1, E(10),S0]
Accion --> Desplaza(Numero,S4)

Pila   --> [+(),S4, Numero(5),S2,+(),S1, E(10),S0)]
Accion --> Reduce(+,1, FuncionAnonima)

Pila   --> [T(5),S2,+()),S1), E(10),S0]
Accion --> Desplaza(T,S3)

Pila   --> [+(),S3,T(5),S2,+(),S1, E(10),S0]
Accion --> Reduce(+,3, FuncionAnonima)

Pila   --> [E (15), S0]
Accion --> Desplaza (E,S1)

Pila   --> [+(),S1, E,(15),S0]
Accion --> Desplaza(+,S2)

Pila   --> [ Numero(3),S2,+(),S1, E (15),S0]
Accion --> Desplaza(Numero,S4)

Pila   --> [EOF(),S4, Numero (3),S2,+(),S1,E(15), S0)]
Accion --> Reduce(EOF,1, FuncionAnonima)

Pila   --> [T(3),S2,+(),S1, E(15),S0]
Accion --> Desplaza(T,S3)
```

```
Pila   --> [EOF(),S3),T(3),S2,+(),S1), E(15),S0)]
Accion --> Reduce(EOF,3 ,FuncionAnonima)

Pila   --> [E(18),S0]
Accion --> Desplaza(E,S1)

Pila   --> [EOF(),S1),E(18),S0]
Accion --> Reduce(EOF,1, FuncionAnonima)

Aceptado  [ini(18),S0]
```

BIBLIOGRAFÍA

[1] http://www.monografias.com/trabajos11/compil/compil.shtml#cla

[2] José Antonio Jimenez Millán, Compiladores y Procesadores de Lenguaje. Publicaciones Universidad de Cádiz: Textos básicos Universitarios: 2004.

[3] Kenneth C. Louden, Construcción de Compiladores. Principios y Prácticas: Thomsom Paraninfo, S.A.

[4] Aho, A.V., Sethi, R. Ullman, J.D., Compiladores Principios, técnicas y herramientas: Addison Wesley Iberoamericana, S.A.: 1990.

[5] http://www.cvc.uab.es/shared/teach/a20364/teoria/tema2.pdf

[6] Sergio Gálvez Rojas, Miguel Ángel Mora Mata, JAVA a tope: Traductores y Compiladores con LEX/ YACC, JFLEX/CUP y JAVA CC. Universidad de Málaga: 2005.

[7] http://www.cvc.uab.es/shared/teach/a20364/teoria/tema3.pdf

[8] G. Sánchez Dueñas, J.A. Valverde Andreu, Compiladores e intérpretes. Un enfoque pragmático. Madrid: ED. Diaz Santos: 1984.

[9] Dick Grune, Henri E.Bal, Ceriel J.H. Jacobs y Koen G. Langendoen, Diseño de compiladores modernos. España: Mc Graw Hill: 2007.

[10] Manuel Alfonseca Moreno, Marina de la Cruz Echeandía, Alfonso Ortega de la Puente, Estrella Pulido Cañabate, Compiladores e intérpretes: teoría y práctica. Universidad Autónoma de Madrid: Pearson Prentice Hall: 2006

ÍNDICE

A

acciones semánticas, 46

análisis ascendente, 19

análisis descendente, 19

analizador léxico, 14

analizador semántico, 46

analizador sintáctico, 16, 31

árbol sintáctico, 16, 18, 46

aristas, 57

atributos, 46

atributos heredados, 49

atributos sintetizados, 48

B

Bison, 4

C

clausura, 37, 55

compilador, 3, 13, 14

Conflicto Desplazar-Reducción, 30

Conflicto Reducción-Reducción, 30

D

derivación, 17, 20

derivación por la izquierda, 20

desplazar, 24

E

expresiones regulares, 15

G

generación de código, 46, 50

gramáticas de atributos, 46

gramáticas libres de contexto, 16

L

LALR (1), 26, 62

lenguaje de alto nivel, 2

lenguaje ensamblador, 2

lenguaje máquina, 2

LEX/ FLEX, 5

Ll

LL (k), 25

L

LR (0), 26, 27

LR (1), 26, 36

LR (k), 25

P

parser, 16

pila, 32

PRIMEROS, 37

producciones, 17

punto, 23

R

reducción, 20, 39

reducir, 24

reglas, 17

S

scanner, 2, 14, 51

símbolo de adelanto, 26

símbolo inicial, 17

símbolos de adelanto, 35, 37
símbolos gramaticales, 17
símbolos no terminales, 17
símbolos terminales, 16, 23
SLR (1), 26

T

tabla de análisis, 31, 64
tabla de símbolos, 14
token, 15
traducción, 13
transición, 29
transiciones, 57

Y

YACC, 3

Firmado: Laia Felip Molina

Bellaterra, 18 de Septiembre de 2009

RESUMEN

El objetivo fundamental de este proyecto consiste en crear un generador de compilador, basado en analizadores ascendentes. Como base para hacer este analizador se usará el lenguaje Cosel y el módulo Com, que es un generador de compiladores basado en analizadores descendentes y que actualmente se está utilizando en las prácticas de la asignatura de Compiladores I. El nuevo generador, que tiene como entrada una gramática, ha de comprobar si es una gramática ascendente LALR (1) y analizar una cadena de entrada de símbolos usando dicha gramática.

RESUM

L'objectiu fonamental d'aquest projecte consisteix en crear un generador de compilador, basat en analitzadors ascendents. Com a base per fer aquest analitzador s'utilitzarà el llenguatge Cosel i el mòdul Com, que és un generador de compiladors basat en analitzadors descendents i que, actualment, s'està utilitzant en les practiques de l'assignatura de Compiladors I. El nou generador, que té com entrada una gramàtica, ha de comprovar si és una gramàtica ascendent LALR (1) i analitzar una cadena d'entrada de símbols utilitzant aquesta gramàtica.

ABSTRACT

The main objective of this project consists of creating a compiler generator based on ascending analyzers. To make this parser, we will use the Cosel language and Com module. This module is a compiler generator based on descending analyzers and is being used in the practice of Compilers I. The new generator, which takes as input a grammar, it has to check if it is a LALR(1) ascending grammar and analyze an input string of symbols using the grammar.